Giancarlo Baccolini

Progetti speciali con il Vic-20 e il C64

Il piacere del computer

Il piacere del computer

1 *Tom Rugg e Phil Feldman* 32 programmi con il PET
2 *Rich Didday* Intervista sul personal computer, hardware
3 *Tom Rugg e Phil Feldman* 32 programmi con l'Apple
4 *Ken Knecht* Microsoft Basic
5 *Paul M. Chirlian* Pascal
6 *Tom Rugg e Phil Feldman* 32 programmi con il TRS-80
7 *Rich Didday* Intervista sul personal computer, software
8 *Herbert D. Peckham* Imparate il Basic con il PET/CBM
9 *Karl Townsend e Merl Miller* Il personal computer come professione
10 *Karen Billings e David Moursund* Te ne intendi di computer?
11 *T. Dwyer e M. Critchfield* Il Basic e il personal computer, uno: introduzione
12 *Don Inman e Kurt Inman* Imparate il linguaggio dell'Apple
13 *T. Dwyer e M. Critchfield* Il Basic e il personal computer, due: applicazioni
14 *Luigi Pierro* Il manuale del CP/M
15 *Carlo Sintini* A scuola con il PET/CBM
16 *David Johnson-Davies* Il manuale dell'Atom
17 *David E. Schultz* Il libro del Commodore VIC 20
18 *Jim Huffman e Robert Bruce* Il "debug" nei personal computer
19 *John M. Nevison* Programmazione in Basic per l'uomo d'affari
20 *Mark Harrison* Imparate il Basic con lo ZX81
21 *Ronald W. Anderson* Dal Basic al Pascal
22 *Herbert D. Peckham* Imparate il Basic con il Texas TI 99/44
23 *Sergio Borsani* A scuola con il Texas TI 99/4A
24 *Jerry Willis e Deborrah Willis* Come usare il Commodore 64
25 *Mark Harrison* Imparate il Basic con lo Spectrum
26 *Carlo Sintini e Costantino Mustacchio* A scuola con il Commodore 64
27 *David A. Lien* Imparate il Basic con l'IBM PC
28 *Ken Tracton* Introduzione al Lisp
29 *Fabio Mavardcchio* Programmi in Basic per l'elettronica
30 *Ian Stewart e Robin Jones* Il linguaggio macchina dello Spectrum
31 *Tom Rugg, Phil Feldman e C.S. Wilson* 32 programmi per il VIC 20
32 *Merl Miller e Mary A. Myers* Introduzione all'Apple Macintosh
33 *Stam Krute* Grafica e suoni con il Commodore 64
34 *Jerry Willis e William Manning* Come usare l'IBM PCjr
35 *Tom Rugg e Phil Feldmann* 32 programmi con il Commodore 64
36 *Sergio Borsani* Matematica e geometria con il Commodore 64
37 *David Laine* ZX Spectrum: tecniche avanzate di linguaggio macchina
38 *Salvatore Marseglia* Chimica con il pocket computer
39 *Patrizio Quintili* Basic per i geometri
40 *Roy Atherton* Programmare in SuperBasic con il QL
41 *Carl Townsend* Il sistema operativo MS-DOS
42 *Carlo Sintini e Costantino Mustacchio* Grafici di funzioni
43 *Salvatore Marseglia* Chimica con il personal computer
44 *Paul Chirlian* Programmare in Forth
45 MSX Basic, guida di riferimento *a cura della Comtrad*
46 *Lawrie Moore* Musica, grafica e programmazione per Spectrum e Spectrum Plus
47 *Giancarlo Baccolini* Progetti speciali con il Vic-20 e il C64
48 *Stanley Trost* Economia e finanza personale in Apple Basic

Giancarlo Baccolini

Progetti speciali con il Vic-20 e il C64

franco muzzio editore

Presentazione

Il volume è indirizzato ai numerosi utenti del VIC-20 della Commodore che ne vogliano sfruttare al massimo le notevoli possibilità; inoltre i concetti esposti valgono in tutta generalità anche per il personal computer C64.
L'autore si è prefisso di condurre gradualmente i lettori a una migliore comprensione della organizzazione interna, sia del software che dello hardware, della macchina, così da permettere un'efficace utilizzazione di tutto ciò che il VIC-20 offre.
Per quanto riguarda il software si analizzano e i modi con cui è gestita la memoria interna e le sua interazioni con l'interprete Basic, per giungere alla descrizione delle principali routine del sistema operativo con esempi del loro uso sia in ambiente Basic che in programmi scritti in linguaggio macchina.
Per quanto concerne l'hardware, oltre alla descrizione dei vari modi di gestione dei dispositivi periferici (registratore a cassette, unità a dischi, plotter e stampante), sono presentate alcune realizzazioni, di interfacce di ingresso e di uscita, che forniscono al lettore i criteri generali di progetto da utilizzare nel caso egli volesse cimentarsi nella costruzione di altri dispositivi periferici.

Indice

CAPITOLO UNO

Organizzazione interna del VIC

In questo capitolo sono elencate le varie unità funzionali che costituiscono il VIC-20 e il loro scopo.

SCHEMA A BLOCCHI

Lo schema a blocchi del VIC-20 è illustrato in figura 1.1. In esso sono indicate le unità funzionali che lo compongono e le vie di comunicazione, o bus, che permettono alle varie unità di scambiarsi tra loro delle informazioni.
Come si può vedere dallo schema, esso è concettualmente diviso in due parti: una relativa al microprocessore 6502, CPU (*Central Processing Unit*), e una relativa al 6561, VIC (*Video Interface Chip*).
La CPU è il cuore dell'intero sistema poiché è l'unico dispositivo che ha il completo controllo di tutte le risorse del calcolatore; esso è in grado di eseguire le istruzioni residenti nelle ROM (*Real Only Memory*, memoria a sola lettura) in cui sono permanentemente memorizzati tutti quei programmi necessari al funzionamento del calcolatore e che lo caratterizzano rispetto ad altri.
Il 6561 è invece preposto essenzialmente alla gestione delle informazioni che sono inviate al modulatore video e che appariranno sul televisore: esso funziona a una velocità superiore a quella della CPU e quindi nello schema di figura 1.1 sono indicati come separati i bus della CPU da quelli del VIC anche se in pratica tale separazione è solo concettuale; si noti che anche il 6561 è asservito al microprocessore nel senso che quest'ultimo

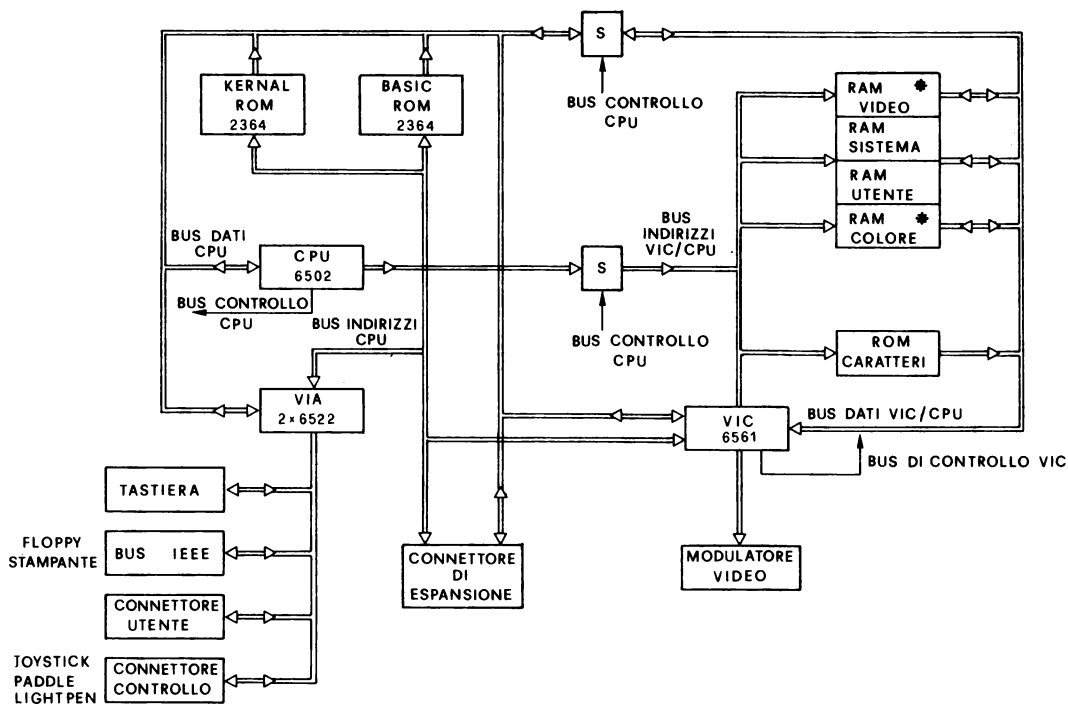



Fig. 1.1 Schema a blocchi del VIC-20.

può imporgli, tra l'altro, i modi di gestione della memoria video e di colore.
Tali memorie sono di tipo non permanente, diversamente dalle ROM, cioè il loro contenuto di informazione va perso se si spegne il calcolatore; in esse la CPU può memorizzare informazioni e rileggerle quando è necessario. Sono memorie ad accesso casuale (*Random Access Memory*); la suddivisione in RAM CPU, RAM video e RAM colore è fittizia e serve solo per ricordare che nell'insieme della RAM si possono sempre individuare delle aree dedicate:

- al sistema
- alla memorizzazione dei caratteri alfanumerici o grafici che appariranno nello schermo televisivo
- al colore di questi ultimi
- ai programmi scritti dall'utente

La prima area, di sistema, merita una descrizione un po' più approfondita in quanto in essa sono memorizzate informazioni utilizzate da:

1. Il sistema operativo, o KERNAL, cioè quel programma, residente nella ROM omonima, che presiede alla gestione delle varie unità costituenti il calcolatore; esso caratterizza il VIC-20 rispetto a un altro tipo di calcolatore: in pratica il KERNAL contiene tutte quelle routine (= programmi) necessarie al funzionamento della tastiera, del 6561, del registratore a nastro, dell'unità a dischi eccetera.
2. L'interprete Basic, cioè quel programma, residente nell'omonima ROM, in grado di accettare le istruzioni in linguaggio Basic e di eseguirle.
3. Il microprocessore stesso in condizioni particolari che possono verificarsi nel corso dell'esecuzione di un qualsiasi programma sia di KERNAL sia di interprete Basic sia di utente.

La parte di memoria RAM utilizzata dal 6561 è, come già detto, quella relativa allo schermo video e al colore: essa contiene informazioni, scrittevi dalla CPU, che, assieme a quelle contenute nella ROM di caratteri, permettono al 6561 di generare l'immagine televisiva.
La RAM di programma è invece dedicata alla memorizzazione dei programmi scritti dall'utente. Programmi che possono essere in Basic, in linguaggio macchina, in Forth eccetera. Essa è gestita o dal KERNAL o dall'interprete Basic ed è utilizzabile solo dal microprocessore.
Per quanto riguarda le ROM, oltre a quella di KERNAL e a quella dell'interprete Basic, è presente quella di caratteri che contiene le informazioni di come "appare" sul video un carattere alfanumerico o grafico.
Le VIA (*Versatile I/O Adapter*) sono dei dispositivi cui si può imporre un certo modo di funzionamento piuttosto che un altro, perciò sono detti programmabili, che permettono di acquisire o di fornire, cioè di scambiare, informazioni con l'esterno del calcolatore. Queste porte di ingresso/uscita sono utilizzate dal KERNAL per acquisire i caratteri dalla tastiera, per comunicare con il registratore a nastro, con la stampante, con joystick, Paddle ecc.
Nello schema a blocchi è indicato anche un connettore di espansione: in esso sono riportati i bus di indirizzo, di dati e di controllo, generati dalla CPU di modo che è possibile aggiungere al calcolatore risorse ulteriori, cioè RAM, ROM, porte di ingresso ecc.

ALLOCAZIONE DELLA RAM

Il microprocessore accede alle varie unità funzionali presenti nel calcolatore dopo avere posto nel bus degli indirizzi l'indirizzo del dispositivo (RAM, ROM, VIA ecc.) interessato allo scambio di informazioni.

Per motivi che appariranno ovvi nei capitoli successivi, conviene che l'utilizzatore del calcolatore conosca i valori di indirizzo associati ai diversi dispositivi.
Questi ultimi infatti non sono individuati sempre dallo stesso indirizzo in quanto alcuni di essi sono diversamente allocati a seconda delle espansioni di memoria RAM o ROM collegate al VIC-20.
Tale allocazione è effettuata automaticamente dal KERNAL al momento della accensione del calcolatore allo scopo di individuare un'area continua di memoria RAM in cui risiederanno i programmi scritti in Basic.
La necessità di un'area senza soluzioni di continuità è imposta dall'interprete Basic il quale non è in grado di eseguire istruzioni in Basic che non siano memorizzate una di seguito all'altra.
A ciò si deve aggiungere che anche il 6561 ha bisogno di un'area di RAM continua di 506 locazioni destinata alla memoria video: questa può iniziare solo dalle locazioni 4096, 5120, 5144 e 7680 (che sono specifiche proprie del 6561).
Come si può notare dalla figura 1.2 nel caso di un VIC-20 senza espansioni di memoria aggiunte l'area destinata alla memorizzazione del programma in Basic va dalla locazione 4096 alla 7679 compresa; l'area di memoria video va dalla 7680 alla 8191 compresa.
Nel caso si aggiunga una espansione di 3K byte si ha a disposizione dei programmi Basic l'area che va dalla locazione 1024 alla 7679 compresa, ferma restando quella dedicata alla memoria video.
Nel caso invece sia collegata una memoria di espansione di 8K byte, che aggiunge memoria dalla locazione 8192 alla 16383 compresa, il KERNAL evidentemente deve allocare l'area Basic in modo diverso per non averla spezzettata in due parti (da 4096 a 7679 e da 8192 a 16383): allora esso predispone le cose indicando al 6561 che l'area di memoria video va dalla locazione 4096 alla 4607 e all'interprete Basic che quella riservata ai programmi va dalla locazione 4608 in poi.
In questo caso è necessario anche un cambiamento dell'area di memoria di colore: questa è fisicamente costituita da 1024 locazioni, di 4 bit ciascuna, che vanno dall'indirizzo 37888 al 38911 compreso; solo 506 sono necessarie al 6561 ma, dato il modo di funzionamento di quest'ultimo, anche qui occorre che il KERNAL "dica" al VIC dov'è l'area effettiva da usare a seconda della espansione di memoria.
In pratica per un VIC-20 senza espansioni o con al massimo quella da 3K byte questa area inizia in locazione 38400; per le espansioni da 8K in su essa inizia in locazione 37888. Ma come fa il KERNAL a rilevare la presenza delle eventuali espansioni aggiunte? La risposta è che esso, dopo l'accensione del calcolatore, scrive un valore, non importa quale,

	LOCATION DEC	HEX	VIC	VIC + 3K	VIC + 8K	VIC + 16K
1K	0 1023	0000 03FF	MICROSOFT BASIC RAM	MICROSOFT BASIC RAM	MICROSOFT BASIC RAM	MICROSOFT BASIC RAM
3K	1024 4095	0400 01FF	NON-EXISTENT (3K EXPANSION RAM)	USER BASIC PROGRAM AREA	NON-EXISTENT (3K EXPANSION)	NON-EXISTENT (3K EXPANSION)
3.5K	4096 7679	1000 1DFF	USER BASIC PROGRAM AREA		SCREEN RAM 4607 4608 USER BASIC PROGRAM AREA	SCREEN RAM 4607 4608 USER BASIC PROGRAM AREA
.5K	7680 8191	1E00 1FFF	SCREEN RAM (VIDEO MATRIX)	SCREEN RAM (VIDEO MATRIX)		
8K	8192 16383	2000 3FFF	NON-EXISTENT (8K EXPANSION ROM/RAM)	NON-EXISTENT		
8K	16384 24575	4000 5FFF	NON-EXISTENT (8K EXPANSION ROM/RAM)	NON-EXISTENT	NON-EXISTENT	
8K	24576 32767	6000 7FFF	NON-EXISTENT (8K EXPANSION ROM/RAM)	NON-EXISTENT	NON-EXISTENT	NON-EXISTENT
4K	32768 36863	8000 8FFF	CHAR ROM (CHARACTER MATRIX)	CHAR ROM (CHARACTER MATRIX)	CHAR ROM	CHAR ROM
	36864 36879	9000 900F	VIC(6561)CHIP	VIC(6561)CHIP	VIC(6561)CHIP	VIC(6561)CHIP
	36880 37135	9010 910F	(VIC CHIP?)	?	?	?
	37136 37151	9110 911F	VIA(6522)CHIPS I/O	VIA(6522)CHIPS I/O	VIA(6522)CHIPS I/O	VIA(6522)CHIPS I/O
	37152 37167	9120 912F	I/O	I/O	I/O	I/O
1K	37888 38399	9400 95FF	Free Nybbles	Free Nybbles	COLOR RAM	COLOR RAM
	38400 38911	9600 97FF	COLOR RAM	COLOR RAM	Free Nybbles	Free Nybbles
2K	38912 40595	9800 9FFF	EXPANSION? I/O NON-EXISTENT	EXPANSION? I/O NON-EXISTENT	EXPANSION? I/O NON-EXISTENT	EXPANSION? I/O NON-EXISTENT
8K	40960 49151	AC00 BFFF	EXPANSION ROM NON-EXISTENT	EXPANSION ROM NON-EXISTENT	EXPANSION ROM NON-EXISTENT	EXPANSION ROM NON-EXISTENT
8K	49152 57343	C000 DFFF	BASIC ROM	BASIC ROM	BASIC ROM	BASIC ROM
8K	57344 65535	E000 FFFF	KERNAL ROM	KERNAL ROM	KERNAL ROM	KERNAL ROM

Fig. 1.2

in ciascuna delle locazioni associate alla memoria RAM e immediatamente ne va a rileggere il contenuto.
È evidente che se quella locazione è di RAM allora quanto letto coincide con quello che era stato scritto; se invece non c'è RAM ciò non accade:

allora il KERNAL riesce ad avere una mappa dell'area effettiva in cui esiste della RAM. L'informazione relativa alla attuale configurazione della memoria, assieme ad altre che vedremo in seguito, è depositata dal KERNAL nella RAM di sistema a disposizione dell'interprete Basic.
Nella figura 1.2 si può notare la presenza di un'area, che va dalla locazione 40960 alla 49151, destinata alle cartucce Commodore, e indicata come area RAM/ROM.
Il lettore si sarà certamente accordo che, se è inserita una cartuccia, per esempio di giochi, gli è impossibile avere il controllo del calcolatore dato che all'accensione il gioco incomincia immediatamente.
In che modo agisce il KERNAL in questo caso? Anche qui esso inizializza alcune variabili nell'area RAM di sistema, ad esempio quelle già viste relative all'area di memoria, e poi va a verificare se è inserita una cartuccia: tale verifica è agevole in quanto nelle locazioni dalla 40964 alla 40968 debbono essere memorizzati nella ROM contenuta in una cartuccia del tipo suddetto i codici corrispondenti alla stringa A0CBM.
Una volta rivelata la presenza di tale stringa il KERNAL cede il controllo del calcolatore non più all'interprete Basic ma a una routine, residente nella cartuccia, il cui indirizzo di partenza è contenuto nelle locazioni 40960 e 40961.
È questa routine che "personalizza" il calcolatore secondo le necessità del gioco. Si deve notare che le routine presenti nel KERNAL sono sempre a disposizione e quindi possono essere utilizzate anche dalle cartucce.
Sempre dalla figura 1.2 si può vedere come vi siano aree dedicate sia ai due VIA che al 6561; ciascuna è costituita solo da 16 locazioni, associate ai vari registri, interni a questi dispositivi, che ne permettono la programmazione.
Esistono altre due aree di notevole importanza per eventuali espansioni di ingresso o di uscita: sono ambedue da 1024 locazioni e iniziano dalla 38912 e dalla 39936 rispettivamente; se ne vedrà una utilizzazione quando si parlerà del convertitore A/D e del programmatore di EPROM.

LE VIA

Il VIC-20 comunica con i dispositivi periferici tramite due circuiti integrati programmabili, VIA (*Versatile Interface Adapter*), i quali gestiscono la tastiera, il registratore, i joystick, la comunicazione seriale di tipo RS 232, il bus seriale IEEE 488.
Le due VIA controllano ciascuna 16 linee di ingresso/uscita (I/O) e quattro linee di *handshaking*.

Il funzionamento di questi circuiti integrati è controllato dal contenuto di 16 registri interni sui quali è possibile effettuare una programmazione, cioè scrivendovi i valori opportuni è possibile, per esempio, far sì che una certa linea sia di ingresso invece che di uscita (o viceversa). Una delle due VIA, il secondo, è dedicato completamente alla gestione della tastiera, mentre il primo è, come si vedrà, quasi completamente a disposizione dell'utilizzatore.
La figura 1.3 dà un'idea di come sono utilizzate le due VIA.
In ogni VIA sono presenti due porte di I/O, dette porta *A* e porta *B*, ognuna corredata da due linee di controllo, CA1 e CA2 per la *A* e CB1 e CB2 per la *B*.
La porta *A* è formata da otto linee, PA7-PA0, le quali possono essere programmate indipendentemente in modo da funzionare come linee o

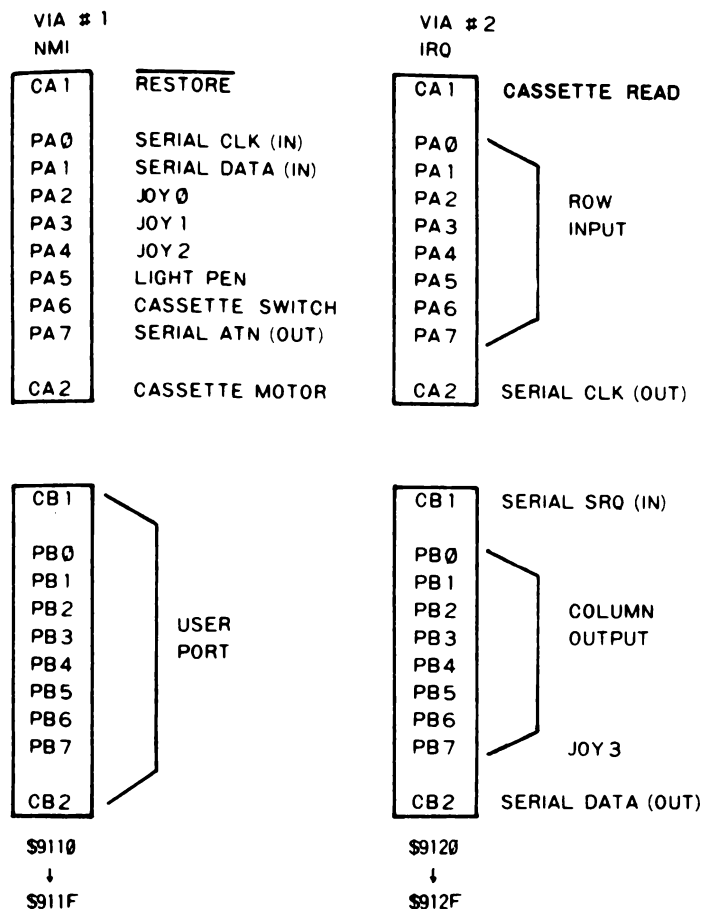



Fig. 1.3 Allocazione delle linee I/O dai due chip 6522.

di ingresso o di uscita. Tale programmazione è attuata scrivendo secondo delle regole precisate dal costruttore un byte opportuno nel registro di direzione dedicato alla porta *A* (DDRA).
I segnali presenti sulle linee di uscita sono imposti dal contenuto del registro di uscita associato alla porta *A*. Il valore presente sulle linee programmate come ingresso può essere memorizzato in un registro interno della VIA se viene presentato un impulso di tensione all'ingresso CA1. Le linee di controllo CA1 e CA2 possono servire come ingressi di interruzione o come linee di *handshaking* per quelle periferiche che necessitano di particolari protocolli di colloquio.
La porta *B* funziona allo stesso modo della porta *A* ed è corredata dagli stessi tipi di registri di controllo; inoltre la linea PB7, programmata come uscita, può essere controllata da uno dei due temporizzatori presenti nella VIA. La linea PB6, se di ingresso, può servire appunto come ingresso di conteggio per l'altro temporizzatore. Le linee di controllo CB1 e CB2, oltre a poter funzionare come le analoghe della porta *A*, possono costituire linee di ingresso o di uscita seriale sotto il controllo del registro a scorrimento presente nella VIA.

IL CONNETTORE DI ESPANSIONE

Il connettore di espansione permette di aggiungere sia memoria sia porte di ingresso o di uscita esterne al VIC-20. Le 44 linee del connettore costituiscono una via di accesso ai bus del calcolatore per i dispositivi aggiunti: in pratica queste linee prolungano all'esterno i bus interni del VIC-20.
Le linee del connettore possono essere raggruppate funzionalmente nei seguenti gruppi:

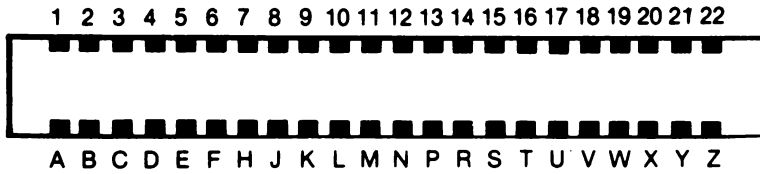
1. linee del bus di dati: CD7-CD0
2. linee del bus degli indirizzi: CA0-CA13
3. linee del bus di controllo: clock di sistema, ingressi di richiesta di interruzione, ingresso di reset, segnali di R/W
4. linee di selezione dei blocchi di memoria e dei dispositivi di ingresso/uscita: /RAM1, /RAM2, /RAM3, /BLK1, /BLK2, /BLK3, /BLK5, /I/02, /I/03
5. linee per l'alimentazione dei dispositivi esterni.

Si può notare dall'elenco che le linee del bus degli indirizzi sono solo 14 e non 16 come ci si potrebbe aspettare: il motivo è che le linee di selezione forniscono già una forma di indirizzamento decodificata e quindi è come

Tabella 1.1

MEMORY EXPANSION

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

A B C D E F H J K L M N P R S T U V W X Y Z

PIN #	TYPE
1	GND
2	CDØ
3	CD1
4	CD2
5	CD3
6	CD4
7	CD5
8	CD6
9	CD7
1Ø	$\overline{\text{BLK1}}$
11	$\overline{\text{BLK2}}$

PIN #	TYPE
12	$\overline{\text{BLK3}}$
13	$\overline{\text{BLK5}}$
14	$\overline{\text{RAM1}}$
15	$\overline{\text{RAM2}}$
16	$\overline{\text{RAM3}}$
17	VR/W
18	CR/W
19	$\overline{\text{IRQ}}$
2Ø	NC
21	+5V
22	GND

PIN #	TYPE
A	GND
B	CAØ
C	CA1
D	CA2
E	CA3
F	CA4
H	CA5
J	CA6
K	CA7
L	CA8
M	CA9

PIN #	TYPE
N	CA1Ø
P	CA11
R	CA12
S	CA13
T	I/Ø2
U	I/Ø3
V	SØ2
W	$\overline{\text{NMI}}$
X	$\overline{\text{RESET}}$
Y	NC
Z	GND

se avessero a disposizione tutti i 16 bit di indirizzo. La tabella che segue indica gli intervalli di indirizzo decodificati dalle linee di selezione assieme ai bit di indirizzo necessari:

Linea di selezione	*Campo indirizzi*	*Bit di indirizzo*
/RAM1	1024-2047	CA9-CA0
/RAM2	2048-3071	CA9-CA0
/RAM3	3072-4095	CA9-CA0
/BLK1	8192-16383	CA12-CA0
/BLK2	16384-24575	CA12-CA0
/BLK3	24576-32767	CA12-CA0
/BLK5	40960-49151	CA12-CA0
/I/02	38912-39935	CA9-CA0
/I/03	39936-40959	CA9-CA0

Ad esempio se si vuole aggiungere una memoria esterna da 8192 byte associata agli indirizzi che iniziano da 40960 occorre attivarne il funzionamento con il segnale /BLK5 e fornirle i bit di indirizzo CA12-CA0. Se invece si vuole aggiungere un VIA esso può essere associato a indirizzi nel campo da 38912 a 39935 oppure nel campo da 39936 a 40959; se vogliamo associare i suoi 16 registri interni a indirizzi che iniziano dalla locazione 39936 esso dovrà essere selezionato dal segnale /I/03, gli si dovranno fornire il segnale di controllo R/W e le linee di indirizzo A0-A3.
La tabella 1.1 mostra la corrispondenza tra i vari contatti del connettore di espansione e i vari segnali che sono stati descritti.

LA PORTA DI UTENTE

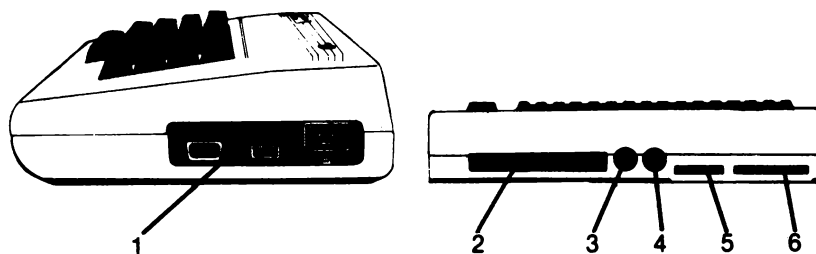
Nel retro del VIC-20 è presente un connettore che costituisce la porta di utente (*user port*); in esso sono riportate le linee della porta *B* del VIA 1, come è indicato nella figura 1.4.
Questa porta è utilizzabile sia come interfaccia generica, di ingresso e/o uscita, per apparecchiature digitali (e a questo scopo è compito dell'utilizzatore programmare opportunamente VIA 1) sia come interfaccia seriale secondo lo standard RS232-C, per la quale fortunatamente sono già implementate nel KERNAL le appropriate routine di gestione.

1). Ingresso I/O per i giochi
2) Espansioni di memoria
3) Audio e Video
4) I/O seriale
5) Cassette
6) Porta di utente

Fig. 1.4 Posizione dei diversi output I/O del VIC.

LA PORTA PER JOYSTICK E PADDLE

Questo connettore riporta all'esterno alcune linee della porta *A* del VIA # 1 e della porta *B* del VIA # 2 e alcune del 6561.
Quelle collegate alle due VIA sono destinate alla lettura della posizione di un joystick attraverso un programma opportuno che deve essere scritto dall'utente.
Per quanto riguarda i collegamenti alle paddle, le relative linee sono direttamente collegate a ingressi dedicati del 6561 relativi a due convertitori analogico-digitale a bassa velocità.
In questo modo è possibile effettuare, tramite un programma opportuno, letture di due registri interni al 6561 stesso i quali contengono ciascuno un valore numerico, compreso tra 0 e 255, il quale corrisponde in modo lineare alla rotazione del potenziometro contenuto in una paddle.
Una linea del connettore in oggetto è utilizzabile per il collegamento con una matita luminosa (*light pen*); questa è un dispositivo che, se puntato in un qualsiasi punto dello schermo televisivo e attivato con la pressione di un pulsante apposito, fornisce un impulso ogni volta che il pennello elettronico del televisore vi passa sotto.
Tale impulso è interpretato dal 6561 come comando per memorizzare in due registri interni al 6561 stesso, e dedicati esclusivamente a tale compito, due numeri relativi alla riga e alla colonna di scansione del pennello elettronico. Si ricordi che essendo proprio il 6561 il dispositivo

che genera i segnali per il televisore, esso è in grado di sapere in ogni istante qual è il punto dello schermo televisivo scandito dal raggio di elettroni.
È allora possibile, con un programma opportuno, acquisire l'informazione su qual è in ogni istante l'areola dello schermo televisivo su cui l'utente punta la matita luminosa (*light pen*).
La matita luminosa è di solito utilizzata in programmi esistenti in commercio che permettono di tracciare disegni a bassa risoluzione; un altro è quello in cui con essa si scelgono opzioni diverse di funzionamento, nei programmi che ne prevedono l'uso.

CAPITOLO DUE

La memorizzazione di programmi in Basic

Per comprendere come si comporta il sistema operativo conviene analizzare quello che succede quando si scrive una linea di programma e si dà il ‹return›.
Il sistema operativo trasferisce la sequenza di caratteri via via introdotti dalla tastiera alla memoria di schermo e quindi, all'attivazione del ‹return›, comprime il testo della istruzione trasformandolo in una sequenza di codici.
Ogni parola chiave del Basic è compressa in un codice, formato da un byte, che la contraddistingue in modo univoco; ogni carattere alfanumerico che fa parte o del nome di una variabile o di una stringa, è invece memorizzato con il suo equivalente in codice ASCII. (Una tabella dei codici delle istruzioni e dei codici ASCII è in appendice 1.)
La sequenza così ottenuta è quindi depositata in un'area di memoria dedicata esclusivamente ai programmi; tutte le sequenze di caratteri o di codici delle successive istruzioni sono memorizzate in locazioni successive di tale memoria.
Ad esempio il programma:

```
10 PRINT "VIC"
20 END
```

nel caso non ci siano espansioni di memoria aggiunte è depositato nell'area di memoria dedicata ai programmi come una sequenza di codici che inizia dalla locazione 4096:

Indirizzo	*codice*
4096	0 (inizio programma
4097	12 (link)
4098	16 (link)
4099	10 (numero di istruzione)
4100	0 (numero di istruzione)
4101	153 (= PRINT)
4102	34 (= virgolette)
4103	86 (V)
4104	73 (I)
4105	67 (C)
4106	34 (= virgolette)
4107	0 (fine istruzione)
4108	18 (link)
4109	16 (link)
4110	20 (numero di istruzione)
4111	0 (numero di istruzione)
4112	128 (= END)
4113	0 (fine istruzione)
4114	0 (fine del programma?)
4115	0 (fine del programma!)

Come si può notare, la prima locazione dell'area di memoria per i programmi contiene il codice di valore 0 che indica proprio l'inizio del programma in Basic. Le successive due locazioni (4097 e 4098) contengono l'indirizzo (link) in cui è memorizzata la prossima istruzione in Basic: tale indirizzo si può calcolare sommando al contenuto della locazione 4097 quello della 4098 moltiplicato per 256. Il valore ottenuto in questo caso è $12 + 16 \times 256 = 4108$; da tale locazione inizia infatti la sequenza di codici dell'istruzione:

```
20 END
```

La locazione 4099 contiene il byte meno significativo del numero di istruzione, la 4100 quello più significativo: in questo caso si tratta della istruzione

```
10 PRINT "VIC"
```

Il codice 153 in locazione 4101 è quello che identifica la parola chiave PRINT.
Le successive cinque locazioni contengono i codici di "VIC". La 4107

contiene 0 e ciò indica al sistema operativo che si tratta della fine dell'istruzione Basic.
In modo analogo è memorizzata l'istruzione

```
20 END
```

si noti che dopo lo 0, che indica la fine di questa istruzione, le successive due locazioni contengono anch'esse 0: ciò è un avviso all'interprete Basic che il programma è terminato e che può tornare a gestire la tastiera per accettare altri comandi dall'operatore.
Da quanto detto dovrebbe risultare chiaro che, agendo opportunamente sulle locazioni di memoria di programma, è possibile, ad esempio, modificare il programma stesso; il programma che segue modifica se stesso la prima volta che viene mandato in esecuzione:

```
10 PRINT "VIC"
20 POKE 4103, ASC("C")
30 POKE 4104, ASC("B")
40 POKE 4105, ASC("M")
50 END
```

Se si dà il RUN a questo programma e poi si fa il LIST, si vedrà che la linea 10 è modificata nella:

```
10 PRINT "CBM"
```

A parte questo esempio banale si possono però utilizzare le informazioni appena descritte per fare cose più interessanti: come si è visto, i primi due byte di ogni sequenza di istruzione contengono l'indicazione della locazione di memoria in cui inizia la (eventuale) prossima istruzione del programma. Sappiamo altresì che se questi due byte contengono ambedue il valore zero, questa è una indicazione all'interprete Basic che si tratta dell'ultima istruzione del programma. Allora se si immette nel calcolatore in modo diretto il comando:

```
PRINT PEEK(4097), PEEK(4098) ‹return›
```

questo scriverà sullo schermo video i valori dei due byte di link (nel caso dei due esempi appena visti scriverà rispettivamente i valori 12 e 16). Se annotiamo questi due valori e poi diamo il comando in modo diretto:

```
POKE 4097, 0: POKE 4098,0 ‹return›
```

nella memoria programmata si avrà a partire dalla locazione **4096**, la sequenza di valori **0 0 0**, che dicono all'interprete Basic che il programma è terminato. Ciò ha come conseguenza che il programma non va in esecuzione né può essere listato anche se risiede ancora in memoria e può essere salvato su nastro o su disco. Chi ha trascritto i due valori di cui sopra è però in grado di ripristinarli, e tramite la:

```
POKE 4097, 12: POKE 4098, 16 <return>
```

rende di nuovo eseguibile il programma caricato da nastro.
Abbiamo appena visto come sfruttando la conoscenza del modo in cui il sistema operativo memorizza il programma in Basic sia possibile scrivere un programma che non può essere utilizzato da chi non conosce la chiave, cioè i valori dei due link nelle locazioni **4097** e **4098**. Questo è però un modo elementare di protezione dei programmi: più avanti saranno descritti metodi più raffinati.
Nell'esempio precedente il programma è memorizzato a partire dalla locazione **4097**: questo indirizzo, assieme ad altri che vedremo tra poco, è automaticamente calcolato dal sistema operativo al momento dell'accensione del calcolatore. È però possibile variarlo quando ciò è necessario, ad esempio, nel caso in cui si voglia riservare una certa area di memoria RAM a programmi scritti in linguaggio macchina.
L'indirizzo di inizio di un programma in Basic è contenuto, nel formato byte meno significativo - byte più significativo, nelle due locazioni di memoria **43** e **44**: in un VIC-20 senza espansioni di memoria il contenuto di queste due locazioni è rispettivamente **1** e **16** ($1 + 16 \times 256 = 4097$). Per spostare l'inizio dell'area dedicata ai programmi basta dare i comandi:

```
POKE 44,X : POKE X × 256,0 : NEW
```

Gli ultimi due sono necessari: il primo per indicare all'interprete Basic dove inizia il programma, il secondo (**NEW**) per far capire al sistema operativo che c'è stata una modifica di indirizzi. Il valore **X** deve essere scelto in base alle locazioni di memoria che vogliamo libere: ad esempio se ne servono 512 basterà che sia **X** = (16 + 2) = 18.
Si è prima accennato alle locazioni **43** e **44**: a partire da queste ci sono delle coppie di locazioni di estrema importanza per l'interprete Basic in quanto contengono gli indirizzi della RAM riservata ai programmi e alle variabili Basic, e delle locazioni di inizio e di fine delle aree ove sono depositate le variabili stesse.
In tabella 2.1 sono elencate le locazioni di cui sopra.
Nel medesimo modo con cui si sono riservate un certo numero di

Tabella 2.1

Indirizzo	*Contenuto*
43 44	indirizzo di inizio di un programma Basic (varia al variare delle espansioni di memoria aggiunte)
45 46	indirizzo di inizio dell'area ove sono depositate le variabili semplici (dipende dalla lunghezza del programma)
47 48	indirizzo di inizio dell'area ove sono depositate le variabili di tipo matrice
49 50	fine dell'area precedente
51 52	fine dell'area ove sono memorizzate le variabili di tipo stringa (ogni volta che si definisce o si modifica una stringa l'indirizzo contenuto in 51 e 52 diminuisce)
53 54	inizio dell'area ove sono memorizzate le stringhe
55 56	indirizzo di fine della memoria Basic

locazioni all'inizio della memoria di programma se ne possono riservare anche alla fine agendo sulle locazioni 51 52 55 e 56: addirittura in questo caso i comandi possono essere scritti come istruzioni all'interno di un programma dato che qui non è necessario dare il comando NEW. Se ad esempio si vuole riservare un'area di 768 (= 3 × 256) locazioni nella parte alta della memoria, occorre dare il comando, o l'istruzione:

(*nnn*) POKE 52, PEEK(52) − 3: POKE 56, PEEK(52) : CLR

in cui si usa CLR, che ha lo stesso scopo del NEW visto nel caso precedente.
È da notare che il contenuto delle locazioni dalla 45-esima alla 54-esima è via via aggiornato dall'interprete Basic e dal sistema operativo man mano che sono immesse istruzioni di un programma o durante la sua esecuzione.

ALLOCAZIONE DELLE VARIABILI

L'area di memoria Basic non utilizzata da un programma è interamente disponibile all'interprete Basic per depositarvi i valori delle variabili relative al programma stesso.
Le variabili possono essere divise in due categorie: quelle semplici, cioè definite da istruzioni come

A = 22 ; A$ = "ABC" ; Z% = 2397

e quelle a più dimensioni, definite da istruzioni DIM:

DIM A(120) ; DIM C%(32) : DIM K$(51)

Per ognuna delle due categorie sono possibili tre tipi: le variabili intere, quelle reali e quelle di tipo stringa.
Le variabili semplici, di qualsiasi tipo esse siano, sono memorizzate subito dopo l'area occupata dal programma Basic a partire dall'indirizzo individuato dal contenuto delle locazioni 45 e 46. La quantità di memoria occupata dipende ovviamente dal numero di variabili presenti nel programma.
Ogni variabile semplice impegna sette byte, dei quali i primi due contengono l'informazione riguardante il tipo di variabile, i restanti contengono il valore a essa associato, nel caso si tratti di variabili intere o reali, oppure l'indirizzo a partire dal quale è memorizzata la stringa nel caso di variabili di tipo stringa.
Il tipo di variabile è individuato dal fatto che i codici ASCII delle prime due lettere del loro nome sono o meno aumentati di un valore 128 ($80) secondo regole predefinite dall'interprete. In tabella 2.2 sono indicate le varie combinazioni possibili.

Tabella 2.2

Tipo	*Nome*	*Byte 1-2*	
intere	A%	$C1 $80	("A"+128;""+128)
	AA%	$C1 $C1	("A"+128;"A"+128)
reali	B	$42 $00	("B";"")
	BB	$42 $42	("B";"B")
stringhe	C$	$43 $80	("C";""+128)
	CC$	$43 $C3	("C";"C"+128)

Per quanto riguarda i restanti cinque byte, anche per essi l'informazione contenuta cambia a seconda del tipo.
Per le variabili semplici di tipo intero il valore associato è contenuto nei byte 3 e 4 ed è espresso in codice binario con il segno indicato dal bit più significativo del byte 3. Ad esempio:

	byte 3	byte 4
A% = 10	$00	$0A
A% = 100	$00	$64

Nel caso di variabili di tipo reale il loro valore è espresso dalla formula:

valore = 2 ↑ (X − 81) * (1 + b46*2 ↑ − 1 + b45*2 ↑ − 2 + ...)

ove la X è il valore del byte 3, mentre b46, b45, b44 ecc. sono i valori dei bit 6, 5 ecc. del byte 4, 5, 6, 7; ad esempio:

	byte3	byte4	byte5	byte6	byte7
A = 101	$87	$4A	$00	$00	$00
A = 10	$84	$20	$00	$00	$00

Infatti:

101 = 2 ↑ (87 − 81) * (1 + 4/8 + 10/128)
10 = 2 ↑ (84 − 81) * (1 + 2/8)

Le variabili semplici di tipo stringa hanno invece il byte 3 che contiene il numero di caratteri componenti la stringa mentre i byte 4 e 5 individuano la locazione di memoria a partire dalla quale è memorizzata la sequenza dei caratteri della stringa stessa; ad esempio

	byte3	byte4	byte5
A$="ABC"	$03	XX	YY

(a partire dall'indirizzo XX + 256 * YY sono presenti i byte di valore $41 (="A"), $42 (="B") e $43 (="C")).

Le variabili di tipo matrice sono memorizzate in un modo un po' più complicato: esiste infatti per ogni tipo di variabile un primo gruppo di byte (*header*) che contiene informazioni riguardanti il nome e il tipo della matrice, il numero degli elementi in ognuna delle dimensioni ed infine una coppia di byte il cui contenuto individua dove inizia un'eventuale altra matrice. A questo primo gruppo di byte seguono ordinatamente i valori degli elementi della matrice. Un header di una matrice a una dimensione è illustrato in tabella 2.3:

Tabella 2.3

byte 1 byte 2	byte 3 byte 4	byte 5	byte 6 byte 7
(nome + tipo)	prossima matrice	# dim.	numero di elementi

Nel caso di matrice a una dimensione l'header occupa solo sette byte, in una a due dimensioni nove e così via aggiungendo ai primi cinque byte tante coppie di byte quanto è il numero di dimensioni.
Quanto detto può essere utile per individuare dove sono memorizzate le variabili che ci interessano ma soprattutto per rendere i programmi Basic un po' più veloci. Per far questo dobbiamo sapere come agisce l'interprete Basic sulle variabili di un programma, ad esempio su:

```
10 A = 10
20 B = 20
30 C = 30
40 D = A * B * C
50 PRINT D
```

L'interprete durante l'esecuzione crea nell'area dedicata alle variabili quattro gruppi consecutivi di 7 byte, per memorizzare i valori di A, B, C e D; quando viene eseguita l'istruzione 50 esso va a sondare dall'inizio tutta l'area destinata alle variabili finché non trova i due byte che individuano la variabile D e quindi esegue PRINT.
A questo punto è immediato pensare che se si modifica la prima istruzione del programma precedente nella:

```
10 D = 0 : A = 10
```

l'individuazione della variabile D avviene in un tempo molto più breve di prima in quanto ora essa è la prima a essere trovata; come conseguenza il programma viene eseguito un po' più velocemente.
Per quantizzare i vantaggi del definire per prime, in un programma, le variabili più usate nel corso dello stesso, proviamo a far eseguire il seguente programma:

```
10 TI$ = "000000"
20 FOR T = 1 TO 10000 : NEXT T
30 PRINT TI/60
```

ove la TI/60 ci dice quanti secondi sono stati necessari per eseguirlo: sono circa 12.
Se ora aggiungiamo alle precedenti le istruzioni:

```
1 A1 = 1 : A2 = 2 : A3 = 3 : A4 = 4
2 A5 = 5 : A6 = 6 : A7 = 7 : A8 = 8
```

e lo mandiamo in esecuzione così modificato si vede che ora impiega

circa 15 secondi; questo aumento di durata è imputabile solo al fatto che ora la variabile T è la nona a essere definita.
Se aggiungiamo l'istruzione:

```
0 T = 0
```

e facciamo eseguire ancora una volta il programma si ritrova il valore iniziale di 12 secondi circa.
La conclusione è che le variabili dei cicli FOR-NEXT e quelle più frequentemente utilizzate nel corso di un programma, conviene siano definite per prime anche con assegnazioni prive di significato.
Un altro miglioramento, sempre riguardante la velocità di esecuzione, si ottiene se si omette, ove possibile, il nome della variabile nelle istruzioni NEXT.
Un altro trucco per rendere più veloci i tempi di esecuzione è quello di *non* fare effettuare operazioni aritmetiche o logiche su dati numerici definiti come costanti; ad esempio:

```
FOR I = 1 TO 1000: PRINT 7 * 9 : NEXT
```

è più lenta di:

```
A = 7 : B = 9 : FOR I = 1 TO 1000: PRINT A * B : NEXT
```

e ciò accade perché nel primo caso l'interprete deve effettuare 1000 volte la conversione dei valori 7 e 9, memorizzati nel programma con i rispettivi codici ASCII, in numeri reali, mentre nel secondo caso la conversione è effettuata solo una volta all'esecuzione delle A = 7 B = 9.

TECNICHE DI MERGE

Molto spesso succede che, mentre si sta immettendo da tastiera un programma nel calcolatore, ci si accorge che si potrebbero utilizzare alcuni sottoprogrammi già scritti e memorizzati su nastro o su disco; se si ha bisogno di un solo sottoprogramma e questo è abbastanza corto ci si adatta a malincuore a ribatterlo da tastiera.
Nel caso si vogliano utilizzare più sottoprogrammi, residenti su nastro o su disco, evidentemente il tempo impiegato per la loro immissione nel calcolatore può diventare molto grande se non si adotta un qualche artificio che ci permetta di caricare dei programmi nell'area Basic senza distruggere quelli che già vi risiedono.

La tecnica per ottenere ciò, usualmente detta di *merge* (fusione, letteralmente) è abbastanza semplice non appena si ricordi come sono memorizzati i programmi del calcolatore.
Nell'esemplificazione seguente supporremo che il programma principale sia già in memoria e che i numeri di istruzione dei sottoprogrammi da aggiungere siano tutti maggiori di quelli del programma principale.
Il metodo per eseguire il merge è relativamente semplice e consiste nell'imbrogliare il sistema operativo e l'interprete Basic facendo credere loro che l'area dedicata alla memorizzazione dei programmi inizi dalla locazione in cui termina il programma principale: a questo punto si effettua il caricamento del sottoprogramma e, una volta concluso, si ripristinano le condizioni iniziali.
In pratica si sostituiscono i due byte di valore 0 che indicano all'interprete Basic che le istruzioni di un programma sono terminate con i due byte che formano il link della prima istruzione del sottoprogramma che si vuole aggiungere.
Allora ci occorre sapere ove termina il programma principale; allo scopo basta far eseguire in modo diretto, la solita:

```
A = PEEK(45) + 256 * PEEK(46) : PRINT A
```

la quale fornisce l'indirizzo in cui inizia l'area di memoria dedicata alle variabili, cioè quello della prima locazione dopo i tre byte di valore 0 che segnalano all'interprete Basic il termine di un programma in Basic, nel nostro caso il programma principale. Si fa poi eseguire, sempre in modo diretto, la:

```
A = A - 2
```

la quale fa sì che ora A contenga l'indirizzo della locazione immediatamente seguente quella nella quale c'è il byte di valore 0 che segnala la fine dell'ultima istruzione del programma principale. A questo punto si *annota* la locazione da cui inizia il programma facendo eseguire:

```
PRINT PEEK(43) + 256 * PEEK(44)
```

(per un VIC-20 senza espansioni di memoria aggiunte il valore fornito è 4097) e si impone che l'inizio dell'area di memoria dedicata al programma sia nella locazione individuata dal valore di A, facendo eseguire sempre in modo diretto:

```
POKE 43,A AND 255 : POKE 44, A/256
```

A questo punto si fa caricare il primo sottoprogramma con un normale LOAD terminato il quale abbiamo nel calcolatore i due programmi uno di seguito all'altro. Dei puntatori 43-44 e 45-46, solo i due ultimi sono correttamente impostati: infatti un eventuale comando di LIST fa apparire solo l'ultimo programma caricato. Occorre a questo punto, ripristinare il valore dei puntatori 43 e 44 e cioè fare eseguire la:

```
POKE 43, 1 : POKE 44, 16
```

(nel caso di un VIC-20 senza espansioni di memoria) oppure, per ogni altro caso:

```
POKE 43, nnnn AND 255 : POKE 44, nnnn / 256
```

dove nnnn è il valore precedentemente annotato. A questo punto il programma può essere salvato così com'è oppure, se si deve aggiungere qualche altro sottoprogramma si ripete il procedimento appena descritto.

TECNICHE DI OVERLAY

Qualche volta si ha la necessità di far caricare e mandare in esecuzione da programma altri programmi: di solito si è costretti a questo a causa della limitata disponibilità di memoria del calcolatore che non è in grado di contenere tutto il programma necessario a un certo scopo.
Un tipico esempio è quello in cui un programma fornisce all'operatore una scelta, un menu, tra diverse opzioni, gestite da altrettanti programmi distinti: una volta che si sia introdotta da tastiera una certa scelta, il programma "menu" si incarica di prelevare dalla memoria di massa e di mandare in esecuzione l'opzione desiderata.
Molto spesso il programma "menu" calcola anche i valori di alcune variabili che si vogliono passare inalterate ai programmi successivamente caricati.
Si possono presentare due casi diversi: quello in cui il programma successivo è meno lungo, cioè occupa meno byte in memoria, o è al massimo della stessa lunghezza del programma "menu", e quello in cui è più lungo.

Primo caso

Nel primo caso, dato che i puntatori nelle locazioni 43-44 e 45-46

rimangono sempre inalterati, i valori delle variabili numeriche restano immutati; ciò si può vedere facendo eseguire, ad esempio, il seguente programma:

```
10 REM PROGRAMMA #1
20 REM È PIÙ LUNGO DEL #2
30 A = 50 : PRINT A
40 LOAD"PROGRAMMA #2" : END
```

e successivamente, dando il RUN, il programma #2:

```
12 REM PROGRAMMA #2
22 PRINT A: END
```

Per quanto riguarda le variabili di tipo stringa definite come costanti in una istruzione di assegnazione, le cose non sono così immediate come si può vedere facendo eseguire il programma:

```
100 REM PROGRAMMA #1$
110 A$ = "ABCDEF"
120 LOAD "PROGRAMMA #2$" : END
```

e successivamente il programma #2$:

```
102 REM PROGRAMMA #2$
112 PRINT A$ : END
```

In questo caso la stringa A$ presentata sul video dal programma #2$ non ha niente a che fare con la stringa "ABCDEF": il motivo di ciò è che alla stringa A$ nel primo programma è associato l'indirizzo, in memoria di programma, da cui inizia la sequenza "ABCDEF" della istruzione 110; quando il secondo programma è caricato in memoria in quelle locazioni viene scritto l'equivalente "tokenizzato" dell'istruzione 112 e quindi va perso il contenuto di A$.
La soluzione a questo problema è molto semplice: si deve fare in modo che A$ sia memorizzata nell'area Basic dedicata alle variabili stringa, cioè nella parte alta della memoria che, nel caso di programmi molto corti, non è influenzata in alcun modo dall'operazione di LOAD. Basta allora modificare la:

```
110 A$ = "ABCDEF"
```

nella:

```
110 A$ = "ABCDEF" + ""
```

In questo modo infatti, quando la 110 viene eseguita nel corso del primo programma, l'interprete Basic crea nell'area di memoria dedicata alle stringhe proprio la **A$** che non è evidentemente alterata dall'operazione di concatenamento con la stringa nulla "".
Il fatto che l'interprete agisca in questo modo è dovuto alla sua peculiare proprietà per cui, ogni volta che deve eseguire una operazione su una variabile di tipo stringa, ne deposita il risultato su un'area diversa: se si fa eseguire l'istruzione **A$ = "A" : B$ = "B" : A$ = A$ + B$** e si va ad analizzare il contenuto dell'area di memoria dedicata alle stringhe si vedrà che sono presenti i caratteri "A" "B" e "AB", cioè l'interprete non fa altro che individuare la nuova **A$** con un diverso puntatore.

Secondo caso

In tale caso si deve procedere in modo diverso a seconda che si voglia o meno passare al secondo programma le variabili calcolate nel primo. Se non si vogliono passare le variabili basta molto semplicemente predisporre i puntatori di fine programma, locazioni **45-46**, ai valori propri del secondo programma (si ricorda ancora che il **LOAD** non modifica le locazioni **43-44** e **45-46**). Allo scopo basta che la prima istruzione del secondo programma sia:

```
nn POKE 45,PEEK(174) : POKE 46,PEEK(175) : CLR
```

Infatti le locazioni **174-175** individuano la locazione ove termina un programma appena caricato da nastro o da disco.
Nel caso invece si vogliano passare al secondo programma, più lungo del primo, i valori delle variabili numeriche (per quelle di tipo stringa valgono le considerazioni già fatte) occorre procedere in modo diverso: bisogna cioè far sì che il primo programma sia artificialmente reso di lunghezza pari al secondo. In questo modo infatti i puntatori **45-46** sono correttamente predisposti per il secondo programma e, cosa più importante, l'area di memoria dedicata alle variabili non viene in alcun modo alterata dal **LOAD**.
Un procedimento che si può adottare è il seguente:

a. si carica in memoria il secondo programma (o uno dei "secondi" programmi).

b. si fa visualizzare e *si annota* il contenuto delle locazioni **45-46** tramite:

PRINT PEEK(45), PEEK(46)

c. se i "secondi" programmi sono più di uno si ripassa al punto (*a*) finché non sono stati analizzati tutti.

Esaurita questa prima parte della procedura si individuano i massimi valori assunti dalle locazioni **45-46** precedentemente annotati al punto (*b*), si carica il primo programma, per intenderci quello di "menu", e si aggiunge immediatamente all'inizio l'istruzione:

0 POKE 45, (massimo dei 45) : POKE 46, (massimo dei 46) : CLR

facendo questo per quanto riguarda l'interprete Basic si è reso il primo programma di lunghezza eguale alla massima delle lunghezze dei "secondi" programmi.

ULTERIORI CONSIDERAZIONI NEL CASO DI UTILIZZO DI DISCHI E DI PROGRAMMI IN LINGUAGGIO MACCHINA

Se si vuole far caricare un programma, scritto in linguaggio macchina e residente su disco, da un programma Basic, sorge il problema che l'interprete Basic, non appena effettuato il caricamento, esegue automaticamente un **RUN** per cui viene ancora mandato in esecuzione il programma in Basic dall'inizio, il quale farà caricare a un certo punto il programma in linguaggio macchina e così via all'infinito: ciò accade nell'esempio che segue:

```
  10 ......
  20 ......
       .
       .
       .
 990 ......
1000 LOAD "NOME" , 8 , 1
1010 ......
1020  .
1030  .
```

Occorre allora fare in modo che la linea 1000 sia eseguita solo una volta così che, caricato il programma NOME, l'esecuzione del RUN automatico farà eseguire tutte le istruzioni fino alla 990 e proseguirà dalla 1010. Questo si può ottenere modificando la 1000 nella:

```
1000 IF A = 0 THEN A = 1 : LOAD "nome" , 8 , 1
```

in questo modo essa la prima volta che viene eseguita fa caricare il programma NOME e pone A = 1; la seconda volta, dato che ora A = 1, non viene più eseguito LOAD il che è quanto si voleva.

UN METODO ANTI-NEW

Qualche volta succede, purtroppo, che ci si dimentica di salvare un lungo programma appena e faticosamente scritto e testato, e di dare il comando NEW.
Se non si fanno eseguire altri comandi o istruzioni non è il caso di disperarsi in quanto l'interprete Basic ha solo posto a zero i due byte di link della prima istruzione e ha reso uguali ai contenuti delle locazioni 43 e 44 rispettivamente quelli delle 45 e 46.
Se si dà un LIST infatti appare sullo schermo il messaggio READY, cioè per l'interprete non risulta esserci alcun programma residente in memoria.
In realtà il programma non è stato distrutto: il problema da affrontare è come riuscire a "risuscitarlo".
Ciò è facilmente attuabile dato che si conoscono sia il modo con cui sono memorizzati i programmi nella RAM del calcolatore, sia quali sono state le modifiche attuate dal comando NEW nella RAM stessa; occorre allora:

1. ripristinare i due byte di link della prima istruzione;
2. imporre i valori corretti nelle locazioni 45 e 46.

Per quanto riguarda il primo punto il valore del link corrisponde all'indirizzo della locazione seguente quella contenente il valore zero, che indica la fine della prima istruzione del programma.
Per ciò che concerne i valori da imporre invece nelle locazioni 45 e 46 questi corrispondono all'indirizzo della locazione immediatamente seguente quella in cui si trova il terzo byte consecutivo di valore zero, quello che indica la fine del programma.
Il ripristino del programma si può attuare dando in modo immediato i seguenti comandi:

(*a*) POKE 45, PEEK(55) : POKE 46, PEEK(56) – 1
POKE 51, PEEK(45) : POKE 52, PEEK(46) : CLR

questi servono per riservare una area minima (256 byte) per le variabili nella parte più alta della memoria disponibile. Se non si facesse così, i due comandi successivi, che ci servono per vedere dove termina la prima istruzione del programma "scomparso", depositerebbero i valori delle variabili in essi definite proprio nel mezzo del programma stesso, rendendolo così irrecuperabile.

(*b*) A = PEEK(43) + 256 * PEEK(44) FOR I = A + 4 TO I + 88:
PRINT –I * (PEEK(I) = 0): WAIT 197,64,64 : NEXT

Questo gruppo di comandi scandisce 88 locazioni dell'area di memoria e visualizza i valori della I, cioè gli indirizzi, in cui si trovano dei byte di valore zero, tra i quali quello che indica la fine della prima istruzione. Si deve prender nota solo del primo valore della I diverso da zero in quanto nel programma scomparso potevano esserci istruzioni molto corte e la scansione effettuata ne indicherebbe tutte le corrispondenti locazioni finali.
La scansione avviene a partire dalla quinta locazione dell'area di memoria riservata ai programmi e questo perché le prime tre locazioni contengono senz'altro uno zero, dato che è stato eseguito un NEW, mentre la quarta potrebbe contenere anch'essa un valore zero, questo relativo al numero d'ordine della prima istruzione del programma scomparso.

WAIT 197, 64, 64

permette di far procedere la scansione, un indirizzo alla volta, con l'attivazione di un qualsiasi tasto.
Il valore di I precedentemente annotato ci serve per ripristinare il link mancante tramite le:

(*c*) POKE A , (I + 1) AND 255
POKE A + 1 , (I + 1) / 256

Per quanto riguarda l'individuazione di dove termina il programma, allo scopo di sapere i valori corretti da mettere nelle locazioni 45 e 46, basta far eseguire le:

(*d*) B = PEEK(45) + 256 * PEEK(46)
FOR I = A TO B: PRINT – I * ((PEEK(I)=0) AND (PEEK(I+1)=0)
AND (PEEK(I+2)=0)):NEXT

e annotare il primo valore di I diverso da zero visualizzato nello schermo.
A questo punto si hanno tutti gli elementi necessari a "risuscitare" il programma; facendo eseguire le:

(*e*) POKE 45, (I + 3) AND 255
POKE 46, (I + 3) / 256

si può di nuovo far listare e salvare il programma.

ANTICOPIA

In questo paragrafo sono descritte alcune tecniche elementari di protezione anticopia per i nastri magnetici: al crescere della complessità delle protezioni introdotte cresce la difficoltà nell'individuarle ma non quella di eliminarle, cioè si rende la vita un po' più complicata a chi volesse copiare un nastro protetto, ma alla fine ci riesce.
Bisogna ricordare inoltre che qualsiasi tipo di protezione è inutile nei riguardi di chi disponga di una apparecchiatura di duplicazione di nastri magnetici.
Per quanto riguarda le tecniche di protezione di programmi su disco, argomento che qui non sarà trattato, esse sono molto più complicate e quindi più difficilmente individuabili: in genere con queste tecniche si immettono volutamente nel disco degli errori, riconosciuti dal DOS che bloccano il funzionamento del drive a meno che il DOS stesso non sia stato modificato mediante cunei sotfware opportuni.
Come è noto nel VIC-20 esistono delle locazioni di memoria dedicate alla gestione del nastro magnetico sia durante un LOAD sia durante un SAVE. Queste locazioni, che vanno dalla 828 alla 1019 compresa, sono destinate a contenere i parametri necessari all'operazione di LOAD o SAVE in corso.
La tabella 2.4 specifica il contenuto di tali locazioni.
Per il nostro scopo le locazioni che interessano sono quelle dalla 833 alla 1019.
Infatti le routine del KERNAL che presiedono alle operazioni di SAVE e di LOAD, anche se mostrano sul video solo i primi 16 caratteri del nome del programma che si sta caricando o trasferendo sul nastro magnetico, in realtà verificano (in LOAD) o trasferiscono su nastro (in SAVE) come nome del file o del programma tutti i 1019 − 833 = 186 caratteri presenti in queste locazioni.

Tabella 2.4

Locazione	*Contenuto*
828 $033C	tipo di file
829 $033D	indirizzo di inizio
830 $033E	del programma
831 $033F	indirizzo di fine
832 $0340	del programma
833 $0341	nome
. .	del
1019 $03FB	file

Si noti che se il nome è, come succede di solito, di lunghezza minore di 186 caratteri, il KERNAL riempie il buffer con dei caratteri "spazio" (il cui codice è 32).
A ciò si aggiunga il fatto che il KERNAL è realizzato in modo tale per cui, durante un LOAD, verifica l'uguaglianza di caratteri presenti sull'header del nastro, solo con quelli eventualmente indicati nella

LOAD "nome programma"

a riprova di ciò basti ricordare che se si dà:

LOAD ‹return›

cioè un LOAD senza specificare il nome, il VIC carica il primo programma che trova sul nastro.
Quanto detto è sufficiente per effettuare una protezione anticopia: infatti, se nel momento di salvare il programma che si vuole proteggere da copiature non autorizzate, si dà un nome più lungo di 16 caratteri e se i caratteri messi in più successivamente verificati durante l'esecuzione del programma stesso, per provocare, ad esempio, un reset del calcolatore corrispondono a quelli originati, si è realizzata una forma elementare di protezione.
Se si salva un programma col nome:

CHR$(5)CHR$(31) + "nome programma"

in cui il nome effettivo è "scrivi in bianco/scrivi in blu/nome programma", al momento di un successivo caricamento il calcolatore darà il messaggio:

LOADING nome programma

dato che i caratteri corrispondenti a CHR$(5) e a CHR$(31), non sono visualizzati sullo schermo: la conclusione di tutto ciò è che l'utilizzatore di tale programma pensa che esso si chiami solo NOME PROGRAMMA e se vuole farne una copia la effettuerà con tale nome.
Nel momento di un successivo caricamento da nastro della copia, il programma automaticamente provocherà un reset del calcolatore se viene dato il RUN.
Per effettuare la protezione bisogna, come è stato accennato, verificare la presenza dei caratteri "fantasma" 5 e 31 e in caso essi non ci siano si fa eseguire un SYS 64802 che inizializza il calcolatore come se fosse stato appena acceso.
Tutto ciò può molto semplicemente essere realizzato con una istruzione del tipo:

```
nnn IF PEEK(833) <> 5 OR PEEK(834) <> 31 THEN SYS 64802
```

ove nnn è il numero di linea.
È da ribadire ancora una volta che questo tipo di protezione, come del resto tutti quelli che seguono, evidentemente non sono validi se chi vuole copiare il programma utilizza una apparecchiatura di duplicazione del nastro invece di effettuare un SAVE.
Il lettore si sarà accorto che la protezione appena vista è facilmente individuabile dato che appare nel listato del programma: è necessario allora renderla invisibile.
Allo scopo si ricordi che il carattere CHR$(20) equivale al comando DELETE il quale cancella un carattere alla sinistra della posizione corrente del cursore; si provi a dare il comando in modo diretto:

```
PRINT "A" + CHR$(20)
```

e si noterà che nello schermo televisivo non si riesce a vedere la stringa "A" dato che essa appare per un tempo brevissimo.
Quanto si è appena verificato permette di non fare apparire nel listato la nostra istruzione di controllo purché essa sia abbastanza breve, e ciò per motivi che appariranno chiari in seguito.
L'istruzione di controllo allora la spezziamo in due parti, che saranno perciò più corte dell'originale, e a ogni istruzione aggiungiamo una REM seguita da un numero di caratteri CHR$(20) pari al numero dei caratteri presenti nella istruzione stessa compresi quelli che ne costituiscono il numero d'ordine.
L'istruzione di controllo è allora divisa nelle:

```
nn IF PEEK(833) <> 5 THEN SYS 64802
kk IF PEEK(833) <> 31 THEN SYS 64802
```

Occorre ora verificare quanti caratteri sarebbero visualizzati durante un LIST nelle due istruzioni: nella prima essi sono 28 poiché ci sono le due cifre (nn) per il numero di istruzione, 2 per IF, 5 per PEEK, 3 per le cifre 833, 2 per le parentesi, 2 per i segni di disuguaglianza, 1 per la cifra 5, 4 per THEN, 3 per SYS e infine 5 per l'indirizzo 64802.
L'istruzione nn deve essere modificata nella:

```
nn IF PEEK(833) <> 5 THEN SYS 64802 : REM
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

(deve essere scritta senza spazi e le X equivalgono a CHR$(20) e sono in numero di 32, dato che si debbono aggiungere i caratteri corrispondenti a :REM).
In modo analogo si deve modificare l'istruzione kk.
Un piccolo svantaggio di tale procedura è che in pratica l'istruzione appare per un brevissimo tempo sullo schermo televisivo dato che il processo di cancellazione inizia non appena il calcolatore invia, durante la generazione del listato, i caratteri CHR$(20); tra l'altro è proprio il numero di linea che permane sullo schermo più a lungo.
Si può rimediare a ciò rendendo l'istruzione di controllo ancora più corta; ad esempio:

```
nn IF PEEK(833) <> 5 THEN NEW : REM xxxxxxxxxxxxxxx
```

che provoca la cancellazione del programma in memoria.
Per effettuare in modo automatico l'immissione dei CHR$(20) al posto delle x si possono utilizzare le istruzioni (poste preferibilmente alla fine del programma stesso):

```
6010 S = PEEK(43) + 256 * PEEK(44)
6020 E = PEEK(45) + 256 * PEEK(46)
6030 FORI = S TO E
6040 IF PEEK(I) = ASC("X") THEN POKE I, 20
6050 NEXT
```

Un altro metodo facilmente implementabile in Basic è quello che utilizza il contenuto del buffer di registratore per depositare dei valori associati a certe variabili; se prendiamo lo stesso nome di programma usato negli esempi precedenti, le istruzioni:

```
nn Z=PEEK(833):REM xxxxxxxxxx
kk ON z goto 300, 400, 500, 600, 700
300 SYS 64802
```

```
400 GOTO 300
500 GOTO 300
600 SYS kk
700 (proseguimento del programma)
```

nel caso il nastro sia quello originale, fanno saltare all'esecuzione della istruzione 700, dato che in 833 è contenuto il valore 5; altrimenti si salta a una delle altre o si ha una segnalazione di errore.
Un metodo migliore di protezione si ottiene mediante delle brevi routine in linguaggio macchina inserite ad hoc in quelle eventualmente presenti nel programma; con queste routine si va a verificare ancora la presenza di caratteri "fantasma" e si mandano in esecuzione delle routine che bloccano il calcolatore nel caso tali caratteri non siano presenti.
È evidente che tale metodo comporta per il copiatore un gravoso lavoro di ricerca delle protezioni in quanto egli deve effettuare il disassemblaggio di tutte le routine per individuare quella di protezione.
Uno dei metodi è eseguire, con il programma da proteggere residente in memoria, i seguenti comandi in modo diretto:

(1) A$ = "nome programma" + " [16 spazi] "

dove "nome programma" deve essere al massimo di 16 caratteri

(2) A$ = LEFT$(A$,16) + "*"

in questo modo si è creata la stringa "nome programma" lunga 17 caratteri e con l'ultimo (*) corrispondente alla chiave.
Si memorizza ora A$ nelle locazioni di RAM a partire dalla 673-ma (le locazioni dalla 673 alla 767 non sono utilizzate né dall'interprete Basic né dal sistema operativo) tramite la:

```
FOR I = 673 TO I + 17 : POKE I , ASC( MID$( A$,I,1)) : NEXT I
```

Si memorizzano, sempre utilizzando dei POKE, a partire dalla locazione 691 i valori:

```
173
178
  2
201
 42
240
```

```
  7
 32
138
255
 96
```

i quali corrispondono alle istruzioni della seguente routine in linguaggio macchina che verifica la presenza del carattere "*" in locazione 690:

```
$02B3    LDA $02B2
$02B6    CMP #$ 2A
$02B8    BEQ $02BD
$02BA    JSR $FF8A
$02BD    RTS
```

Si mandano in esecuzione in successione le routine SETNAM,SETLFS e SAVE (si veda il capitolo 7) tramite i comandi in modo diretto:

```
POKE 780 , 28 : POKE 781 , 673 AND 255 : POKE 782 , 673/256 : SYS
65469 : REM SETNAM

POKE 780 , 1 : POKE 781 , 1 : POKE 782 , 255 : SYS 65466 : REM
SETLFS

POKE 780 , 43 : POKE 781 , PEEK(45) : POKE 782 , PEEK(46) : SYS
65496 : REM SAVE
```

a questo punto il programma è salvato col nome composto dal nome vero assieme alla routine di protezione.
Si ricordi che nel programma Basic deve essere presente un SYS 691 , o meglio, nelle routine in linguaggio macchina almeno un JSR $0283 per effettuare il test sulla "originalità" del programma.

UN METODO DI SCRAMBLING

Un metodo elementare di protezione dei programmi in Basic è quello che fa uso della tecnica di *scrambling*: con essa sono modificati, secondo una regola ben precisa, tutti i byte relativi a un programma.
In questo modo il programma stesso evidentemente non è più eseguibile e il listato è incomprensibile.
Il programma è tuttavia ancora salvabile su nastro o su disco di modo

che chi ne conosce la regola con cui è stato effettuato lo scrambling può ricaricarlo in memoria e renderlo di nuovo eseguibile.
La regola più semplice per effettuare questo tipo di protezione è di modificare ogni locazione dell'area di memoria riservata al programma in modo che il byte in essa contenuto sia trasformato nel suo complemento a 255, cioè se il byte vale x sia modificato in $255 - x$.
Per l'operazione inversa, cioè ripristinare i valori iniziali, si effettua la medesima operazione dato che $255 - (255 - x) = x$.
Nell'ipotesi di avere già il programma da proteggere residente in memoria, per ottenere lo scrambling basta far eseguire, in modo immediato, le seguenti istruzioni:

(*a*) A = PEEK(43) + 256 * PEEK(44) : B = PEEK(45) + 256 * PEEK(46)

(*b*) FOR I = A TO B – 3 : POKE I, 255 – PEEK(I) : NEXT

Si noti che la (*b*) fa sì che siano modificati tutti i byte del programma ad eccezione di quelli che indicano la fine del programma stesso, e ciò allo scopo di poterlo salvare su nastro o su disco con il solito SAVE "nome programma".
Nel caso si sia appena caricato in memoria un programma che è stato protetto nel modo descritto in precedenza se si vuole renderlo di nuovo eseguibile basta ancora agire in modo diretto secondo i punti (*a*) e (*b*).
Le regole di scrambling evidentemente possono essere più complicate: ad esempio si potrebbero usare, al posto delle precedenti, le istruzioni:

(*c*) A = PEEK(43) + 256 * PEEK(44): B = PEEK(45) + 256 * PEEK(46)

(*d*) FOR I = A TO B – 3 : POKE I,(250 AND PEEK(I)) + 15 – (PEEK(I) AND 15)

CAPITOLO TRE

La gestione dei dispositivi periferici

In questo capitolo si vedrà come il VIC gestisce le comunicazioni con i dispositivi periferici, cioè con il registratore a cassette, la stampante, l'unità a dischi, la tastiera, lo schermo video ecc.
Le periferiche possono essere di tre tipi: quelle solo in grado di inviare informazioni verso il VIC, quelle solo in grado di riceverne, e quelle capaci sia di ricevere che di trasmettere. Al primo tipo appartiene la tastiera, al secondo lo schermo video e la stampante, al terzo l'unità a dischi e il registratore.
A ogni dispositivo è associato un numero, secondo quanto indicato nella tabella 3.1.

Tabella 3.1

Dispositivo	*Numero*
Tastiera	0
Registratore	1
Porta seriale	2
Schermo video	3
Stampante, plotter	4-7
Disk drive	8-11
Altri	12-255

Lo scambio di informazioni tra il VIC e le periferiche avviene tramite

delle porte di ingresso/uscita programmabili la cui gestione è compito del sistema operativo.
Mentre le operazioni di lettura o di salvataggio di programmi, tramite le istruzioni LOAD e SAVE, sono effettuate automaticamente dal VIC-20, la gestione ad esempio della stampante può avvenire solo attraverso opportune istruzioni (OPEN#, GET#, INPUT#, PRINT#).
Il comando OPEN fa sì che il calcolatore apra un canale di comunicazione con un dispositivo periferico. La sintassi di OPEN è

OPEN FL, ND, IS, "comando"

FL sta per "file logico" ed è un numero utilizzato per contrassegnare quel canale rispetto ad altri eventualmente aperti: ogni operazione che riguarda quel canale sarà sempre individuata da quel numero.
Il parametro ND, numero di dispositivo, serve invece per assegnare quel canale alla comunicazione con quel particolare dispositivo: ad esempio

OPEN 5, 1, IS, "nome"

permette di assegnare il canale 5 al registratore: infatti il numero di dispositivo 1 corrisponde proprio al registratore (tabella 3.1).
L'indirizzo secondario IS è anch'esso un numero che specifica il tipo di scambio di informazioni cui la periferica è interessata. Tipici valori di IS sono elencati nella tabella 3.2.

Tabella 3.2

Dispositivo	*IS*	*Funzione*
Registratore	0	lettura di un file
Registratore	1	scrittura di un file
Registratore	2	scrittura di un file + EOT
Stampante	0	selezione dei caratteri maiuscoli-grafici
Plotter 1520	2	selezione dei colori
Disk drive	15	accesso al canale dei comandi

Infine l'ultimo parametro, "comando", è diversamente interpretato a seconda della periferica scelta.

IL REGISTRATORE A CASSETTE

È possibile scrivere sul registratore a cassette non solo programmi ma anche dati.
Per quanto riguarda i programmi si utilizza di solito il comando SAVE e il sistema operativo si incarica di effettuare in modo automatico l'apertura del file logico necessario.
Il comando SAVE può essere di quattro tipi:

SAVE "nome", 0
SAVE "nome", 1
SAVE "nome", 2
SAVE "nome", 3

Il primo tipo, in cui di solito è omesso lo zero e per il quale non è strettamente necessario neanche fornire il "nome", salva il programma Basic assieme a due puntatori che indicano l'inizio dell'area di memoria RAM da cui il salvataggio viene effettuato.
Se invece il SAVE è dato con indirizzo secondario 1 in un successivo caricamento del programma del registratore in memoria, esso sarà posto in RAM a partire dalla locazione data dai due parametri di cui sopra e indipendentemente dalla presenza o meno di eventuali cartucce di espansione RAM.
Il SAVE "nome", 2 fa sì che la registrazione del programma abbia alla fine un carattere speciale di fine nastro, (End Of Tape, EOT); tale carattere, se viene letto dal calcolatore durante il caricamento del programma in memoria, fa sì che venga inviato al video il messaggio:

DEVICE NOT PRESENT

che indica la fine del nastro se per caso non è stato trovato (e caricato) il programma voluto.
Infine la quarta possibilità permette di non far rilocare il programma letto e di far scrivere l'EOT come ultimo carattere.

IL DISK-DRIVE 1541

Il 1541 dà una nuova dimensione al calcolatore cui viene collegato, poiché fornisce una memoria di massa di circa 170.000 byte, con elevata velocità di trasferimento di dati da e verso il calcolatore stesso.

Tabella 3.3 Specifiche del disk drive 1541

Capacità di memoria	174848 byte per disco
Massimo numero di nomi nella directory	144 per disco
Settori per traccia	17-21
Byte per settore	256
Tracce	35
Settori	683 (664 liberi per l'utente)

Il 1541 è un completo sistema a microprocessore di potenzialità pari a quella di un VIC-20 o di un C-64: nel 1541 sono infatti contenuti:

- un microprocessore 6502;
- una memoria ROM, di 16536 byte, che costituisce il sistema operativo del drive stesso (DOS, *Disk Operating System*);
- 2048 byte di memoria RAM;
- 2 porte di ingresso/uscita (I/O) programmabili del tipo 6522;
- tutta l'elettronica di supporto agli attuatori elettromeccanici.

La flessibilità del drive permette un accesso ai dati di tipo causale, a differenza del nastro magnetico in cui l'accesso è strettamente sequenziale. La velocità di trasferimento di dati è inoltre di 7-10 volte superiore a quella del registratore.
Le specifiche del 1541, relative alla capacità di memoria, sono riportate nella tabella 3.3.

SALVATAGGIO E CARICAMENTO DI PROGRAMMI DA DISCO

Queste operazioni si attuano con le stesse modalità già viste nell'utilizzo del registratore, con un più alto grado però di flessibilità, dato che il 1541 è governato da un microprocessore.
Il salvataggio di un programma avviene con la solita:

```
SAVE "nome programma", 8, X
```

ove X può valere o 0 oppure 1.
Il valore di X = 0, che può anche essere omesso, precisa che si vuole che il programma, all'atto del successivo caricamento sul calcolatore, sia allocato nell'area Basic, ovunque essa sia (ciò dipende dalle eventuali espansioni di memoria utilizzate).

Il valore di X = 1 fa sì invece che il programma sia allocato nella medesima area di RAM da cui era stato salvato.
In pratica si tratta delle prime due opzioni per il SAVE già viste con l'uso del registratore.
Esiste però la possibilità di salvare un programma col nome di uno già esistente nel disco; il comando è ora:

SAVE " @0:nome programma", 8, X

ove @0 specifica al DOS il tipo di salvataggio da fare.
Questo comando è utilizzato soprattutto quando si fanno correzioni o modifiche a un programma già memorizzato su disco e si vuole salvare la versione corretta senza cambiarne il nome. Se si utilizzasse il registratore si sarebbe costretti a posizionare il nastro esattamente all'inizio del vecchio programma ed effettuare la registrazione della versione corretta (col pericolo di cancellare il programma successivo registrato nel nastro nel caso la nuova versione fosse più lunga dell'originale).
Nel nostro caso è il DOS che effettua tale sostituzione di versione del programma.
Anche per la verifica di identità tra un programma esistente nel calcolatore e uno che si trova nel disco si usa:

VERIFY "nome programma ",8,X (X = 0, 1)

Qui però occorre fare attenzione perché la verifica, pur con programmi perfettamente identici, può dar luogo al fatidico:

VERIFY ERROR
READY

se il programma è stato caricato su un calcolatore con espansioni di memoria diverse da quelle presenti quando era stato memorizzato sul disco. Ciò è dovuto al fatto che, essendo stata salvata, tramite il SAVE, l'area di memoria Basic, sono state salvate anche tutte le coppie di byte di link delle istruzioni; con una diversa configurazione delle espansioni di memoria, anche se il programma è correttamente caricato e viene eseguito perfettamente, i link hanno valori diversi da quelli originali. Allora l'operazione di verifica, dato che è attuata byte per byte, rileva la differenza tra la copia di programma esistente nel calcolatore e quella residente sul disco: ciò genera il messaggio di errore.
Il DOS permette di usare il carattere "*" al posto del nome del programma nelle istruzioni SAVE, VERIFY e LOAD, con lo speciale significato

di “stesso nome” del programma nominato per ultimo in uno dei suddetti comandi; ad esempio le due successive operazioni sul disco:

```
LOAD "VIC", 8
VERIFY "*", 8
```

fanno sì che l'ultima equivalga a:

```
VERIFY "VIC", 8
```

per quanto riguarda il comando LOAD esso può articolarsi in vari modi:

```
LOAD "nome", 8, X
LOAD "*", 8, X
```

con gli stessi significati per la X già visti nel caso del nastro.
È anche possibile la:

```
LOAD "PI*", 8, X
```

il cui significato è: carica il primo programma sul disco che ha il nome che inizia con le lettere PI.
È ammesso anche:

```
LOAD "??PI?", 8, X
```

che fa in modo che sia caricato in memoria il primo programma sul disco che ha un nome di cinque lettere delle quali la terza e la quarta sono P e I rispettivamente.
È permesso l'uso contemporaneo dei simboli ? e *: ad esempio il comando:

```
LOAD "?AR*", 8, X
```

carica nel calcolatore il primo programma sul disco che ha la seconda e la terza lettera del nome uguali ad A e R rispettivamente, qualsiasi sia la lunghezza del nome del programma e qualsiasi siano le rimanenti lettere che compongono il nome stesso.
Il lettore si sarà certamente accorto che negli esempi precedenti si è sempre detto “il primo programma sul disco”: ma perché proprio il primo e non altri?
La risposta a questa domanda è immediata se si conosce il modo con cui il DOS organizza la memorizzazione delle varie informazioni nel disco.

ORGANIZZAZIONE DEL DISCO

Nel disco, al momento della sua "formattazione", di cui si parlerà più avanti, vengono realizzate dal DOS delle tracce (*tracks*), cioè delle piste magnetizzate, ognuna delle quali è suddivisa in più settori (*sectors*). Il DOS del 1541 genera nel disco 35 tracce, ognuna con un numero di settori, detti anche blocchi, variabile da traccia a traccia da un minimo di 17 a un massimo di 21, in questo modo:

Tracce	*settori*
1-17	21
18-24	20
25-30	18
30-35	17

In un disco sono presenti 683 blocchi, di cui 664 disponibili per la memorizzazione di programmi o di file di dati. La traccia 18 contiene nel suo settore numero zero la mappa dei settori disponibili, detta BAM (*Block Availability Map*), cioè una mappa che indica l'occupazione o meno di ognuno dei settori del disco. Il listato di figura 3.1 fornisce il contenuto di una BAM; sono indicati i valori esadecimali dei 256 byte in essa contenuti e a fianco i caratteri corrispondenti, se stampabili.

```
TRACK 18  SECTOR 0

00 :12 01 41 00 15 FF FF 1F 15 FF FF 1F 15 FF FF 1F :  A   ππ   ππ   ππ
10 :15 FF FF 1F 15 FF FF 1F 15 FF FF 1F 15 FF FF 1F : ππ   ππ   ππ   ππ
20 :15 FF FF 1F 15 FF FF 1F 15 FF FF 1F 15 FF FF 1F : ππ   ππ   ππ   ππ
30 :15 FF FF 1F 15 FF FF 1F 11 D7 5F 1F 00 00 00 00 : ππ   ππ   O←
40 :00 00 00 00 00 00 00 00 10 EC FF 07 00 00 00 00 :           ■π
50 :00 00 00 00 12 BF FF 07 13 FF FF 07 13 FF FF 07 :      ▘■π  ππ   ππ
60 :13 FF FF 07 12 FF FF 03 12 FF FF 03 12 FF FF 03 : ππ   ππ   ππ   ππ
70 :12 FF FF 03 12 FF FF 03 12 FF FF 03 11 FF FF 01 : ππ   ππ   ππ   ππ
80 :11 FF FF 01 11 FF FF 01 11 FF FF 01 11 FF FF 01 : ππ   ππ   ππ   ππ
90 :31 35 34 31 54 45 53 54 2F 44 45 4D 4F A0 A0 A0 :1541TEST/DEMO
A0 :A0 A0 5A 58 A0 32 41 A0 A0 A0 A0 00 00 00 00 00 :  ZX 2A
B0 :00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :
C0 :00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0 :00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0 :00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0 :00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 3.1 Copia di BAM.

Nella traccia 18, settore 1, inizia invece l'indice (*directory*) del disco, in cui il DOS mantiene aggiornato l'elenco di tutti i programmi e dei file memorizzati sul disco.
Il DOS aggiunge in sequenza alla directory i nomi dei successivi file man mano che questi sono salvati sul disco. Il listato di figura 1.2 è un esempio del contenuto del settore 1 della traccia 18.

```
TRACK 18 SECTOR 1

00 :12 04 82 11 00 48 4F 57 20 54 4F 20 55 53 45 A0 :  ▮ HOW TO USE
10 :A0 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 0D 00 :
20 :00 00 82 11 03 48 4F 57 20 50 41 52 54 20 54 57 :  ▮ HOW PART TW
30 :4F A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 05 00 :O
40 :00 00 82 11 09 56 49 43 2D 32 30 20 57 45 44 47 :  ▮ VIC-20 WEDG
50 :45 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 04 00 :E
60 :00 00 82 13 00 43 2D 36 34 20 57 45 44 47 45 A0 :  ▮ C-64 WEDGE
70 :A0 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 01 00 :
80 :00 00 82 13 01 44 4F 53 20 35 2F 31 A0 A0 A0 A0 :  ▮ DOS 5.1
90 :A0 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 04 00 :
A0 :00 00 82 13 03 43 4F 50 59 2F 41 4C 4C A0 A0 A0 :  ▮ COPY/ALL
B0 :A0 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 0B 00 :
C0 :00 00 82 13 09 50 52 49 4E 54 45 52 20 54 45 53 :  ▮ PRINTER TES
D0 :54 A0 A0 A0 A0 00 00 00 00 00 00 00 00 00 09 00 :T
E0 :00 00 82 10 00 44 49 53 4B 20 41 44 44 52 20 43 :  ▮ DISK ADDR C
F0 :48 41 4E 47 45 00 00 00 00 00 00 00 00 00 04 00 :HANGE
```

Fig. 3.2 Copia di directory.

Il carattere "*" usato nei precedenti comandi fa sì che il DOS effettui una operazione di verifica, caricamento o salvataggio relativa al primo programma trovato nella directory che soddisfa alle specifiche date per il nome; se viene impartito il comando:

LOAD "MA*", 8

e nella directory sono elencati di seguito i nomi CAIO, TIZIO, MARTE, MARIO sarà caricato il programma MARTE e non MARIO.
Se invece è dato il:

LOAD "??RI*", 8

sarà caricato il programma MARIO perché è il primo nella directory la cui terza e quarta lettera coincidono con quelle indicate nel "nome" della LOAD.

FORMATTAZIONE DEL DISCO

Quanto finora detto è sufficiente per caricare programmi già presenti su un disco e per salvarli su un disco già formattato.
Se si acquista un disco vergine in esso evidentemente non esistono né la BAM né la directory.
Occorre quindi formattarlo, cioè far sì che il DOS vi generi i blocchi menzionati. Allo scopo è necessario aprire un canale di comunicazione col DOS per impartirgli appropriati comandi.
Lo scambio di informazioni tra il calcolatore e il DOS può avvenire solo con un indirizzo secondario 15 e con numero di dispositivo 8.
Inoltre dato che al 1541 può essere collegato sia un VIC-20 che un C-64, conviene avvisare il DOS su quale di questi due calcolatori il 1541 ha a che fare. Ciò si ottiene molto semplicemente con il comando in modo immediato:

```
OPEN 15, 8, 15,"UI-"
```

per il VIC e con:

```
OPEN 15, 8, 15,"UI+"
```

per il 64. Il primo dei due permette al VIC-20 di comunicare col disco a velocità superiore rispetto al 64.
In questo modo si è aperto un canale di comunicazione, il # 15, per il colloquio col DOS: attraverso questo canale si possono impartire dei comandi al 1541 e si possono leggere eventuali messaggi di errore forniti dal DOS in risposta a comandi errati.
Per formattare un disco occorre dare il:

```
PRINT # 15, "N0:nome,id"
```

dove "N0:" costituisce il comando di formattazione, "nome" è il nome che si assegna a quel disco, e "id" è un identificatore costituito da due caratteri alfanumerici che servono al DOS stesso per distinguere tra loro i vari dischi.
È necessario che i diversi dischi che si utilizzano abbiano un identificatore diverso per evitare spiacevoli sorprese specialmente nel caso si legga un programma da un disco e lo si salvi su un altro: se l'identificatore del secondo disco è lo stesso del primo, il DOS ritiene che i settori liberi siano quelli del primo disco e quindi salva il programma su settori che

potrebbero in realtà essere già occupati; in questo modo si ha la distruzione di quanto vi era contenuto.

ALTRI COMANDI PER IL DOS

Il DOS permette di leggere la directory per mezzo della:

LOAD "$", 8

che fa sì che venga caricata nel calcolatore, come se fosse un programma, la lista dei nomi dei file esistenti nel disco, assieme al loro tipo e al numero di settori occupati da ciascun file.
Se si effettua il LIST di tale "programma" appare sul video qualcosa di simile a quel che si vede in figura 3.3.

```
LIST DI UNA DIRECTORY

0 "1541TEST/DEMO     " ZX 2A
13    "HOW TO USE"        PRG
5     "HOW PART TWO"      PRG
4     "VIC-20 WEDGE"      PRG
1     "C-64 WEDGE"        PRG
4     "DOS 5.1"           PRG
11    "COPY/ALL"          PRG
9     "PRINTER TEST"      PRG
4     "DISK ADDR CHANGE"  PRG
4     "DIR"               PRG
6     "VIEW BAM"          PRG
4     "CHECK DISK"        PRG
14    "DISPLAY T&S"       PRG
9     "PERFORMANCE TEST"  PRG
5     "SEQUENTIAL FILE"   PRG
13    "RANDOM FIAL"       PRG
558 BLOCKS FREE.

READY.
```

Fig. 3.3 Listato di directory.

Come si può notare, dopo ogni nome di file esistono tre possibili estensioni o etichette associate al nome: PRG, SEQ, USR.
Esse stanno a indicare che il file è rispettivamente un programma, un file sequenziale, un file di tipo diverso dai due precedenti. Questa

indicazione può essere utile per etichettare in modo diverso file anche riferentisi a programmi: ci si può infatti trovare nella necessità di salvare un programma in Basic, e il DOS lo salverà automaticamente con estensione PRG; oppure un programma in linguaggio macchina che l'utilizzatore vuole etichettare con estensione USR.
Le possibilità del DOS non si fermano qui; è possibile infatti:

- cambiare il nome di un file di qualsiasi tipo (SEQ, PRG, USR);
- cancellare un file;
- appendere uno di seguito all'altro più file (tipicamente SEQ o USR).

Nella ipotesi che sia già aperto il canale di comunicazione col DOS, il comando:

PRINT# 15, "R0:nuovo nome=vecchio nome"

ad esempio:

PRINT# 15,"R0:CAIO=TIZIO"

permette di dare il nome CAIO al file TIZIO, cioè nella directory non appare più il nome TIZIO ma CAIO.
Invece:

PRINT# 15,"S0:nome"

ad esempio:

PRINT# 15,"S0:TIZIO"

cancella il nome TIZIO dalla directory e quindi è come se quel file non esistesse più. (In realtà si rendono disponibili per altri successivi file i settori già occupati da TIZIO: se non si fanno ulteriori operazioni di salvataggio su disco è possibile rileggere il file cancellato eseguendo immediatamente il comando LOAD "*" , 8.)
Come accennato è possibile anche appendere uno di seguito all'altro più file. Ciò si può ottenere con il comando:

PRINT# 15,"C0:UNO= 0:DUE,0:TRE"

il quale fa sì che il file UNO sia costituito dai file DUE e TRE.
Questa operazione ha senso solo per file di tipo USR o SEQ e non per i programmi: si ricordi infatti che nel salvataggio di un programma sono

memorizzati nel disco i tre byte di valore 0 che indicano la fine del programma stesso. L'accodamento effettuato dalla "C0:.......", mantiene inalterati questo gruppo di byte, per cui all'atto di un successivo caricamento in memoria del file UNO il calcolatore effettua correttamente il LOAD ma quando si dà il RUN del programma esso si arresta alla fine del programma DUE.
Esistono altri due comandi di utilità del DOS: quello di *inizializzazione* del drive e quello di *validazione*.
Il primo è necessario per ripristinare correttamente il DOS nel caso sia stato inviato un comando errato; l'inizializzazione si attua con:

```
PRINT# 15,"I0"
```

Il secondo comando, di validazione, è di estrema utilità, ma va usato con attenzione. Infatti, dopo che un disco è stato utilizzato per un certo tempo, per cui sono stati presumibilmente eseguiti dei comandi di cancellazione o non sono stati chiusi correttamente dei file, la BAM è molto disordinata. Cioè per il DOS risultano ancora occupati dei settori che in realtà sono liberi: ci si può rendere conto di ciò per il fatto che la somma del numero dei settori occupati dai singoli file e dei settori liberi non è pari a 664.
Il comando:

```
PRINT# 15,"V0"
```

permette al DOS di riorganizzare sia la BAM che la directory e quindi di rendere disponibili tutti i settori liberi.
Esiste un pericolo se si usa il comando di validazione, pericolo che per fortuna non riguarda né file di tipo PRG né file SEQ o USR ma solo file di tipo random per i quali è stato necessario informare il DOS, tramite opportuni comandi (che qui non saranno trattati), quali sono i blocchi occupati da questi file. Si deve notare che però i file random non sono memorizzati con alcun nome nella directory e perciò i settori da essi occupati sono considerati liberi nel momento dell'esecuzione del comando "V0".

IL «DOS WEDGE»

Nel disco fornito assieme al 1541 esistono due programmi, il VIC-WEDGE e il C-64 WEDGE, di estrema utilità. Caricati nel calcolatore

e mandati in esecuzione essi, al prezzo di qualche centinaio di byte occupati, forniscono un certo numero di comandi molto utili.
Questi, costituiti da un solo carattere ed eseguiti solo se compaiono nella prima colonna di un comando dato in modo diretto, sono @ e /.
Il comando @ deve essere dato nella forma:

@ argomento

ed equivale alle:

```
OPEN 15, 8, 15
PRINT# 15,"argomento"
CLOSE 15
```

Quindi, ad esempio:

@ S0:PIPPO

equivale alle:

```
OPEN 15, 8, 15
PRINT# 15,"S0:PIPPO"
CLOSE 15
```

Invece

@ I

è il comando di inizializzazione del DOS.
Il comando:

@ $

fa sì che sia listata sul video, *senza però interferire con eventuali programmi che risiedono sul calcolatore*, la directory del disco.
Il comando

@ $:CAI*

visualizza tutti i nomi dei file della directory che iniziano con i caratteri CAI.
Il comando / è equivalente a un LOAD da disco:

/ nome

equivale a

LOAD "nome", 8, 0

In pratica per i nomi dei file, nei comandi del Wedge, valgono le convenzioni già viste per i simboli ? e *.
Se si è appena inizializzato il DOS, il comando:

LOAD "*", 8

fa sì che sia caricato il primo programma elencato nella directory.
Data l'estrema utilità del DOS-Wedge conviene allora che proprio questo programma sia il primo salvato su disco per poterlo mandare in esecuzione con la sequenza di comandi:

OPEN 15, 8, 15, "UI-"
LOAD "*",8

Sono anche ammessi comandi del tipo:

@ $:*=PRG
@ $:*=SEQ
@ $:*=USR

con i quali sono presentati sullo schermo video solo i nomi dei file nella directory rispettivamente di tipo programma, sequenziale o USR.
È anche possibile il comando:

@ $:??MA*

con il quale sono listati nel televisore solo i file che soddisfano alla ??MA*.

LA STAMPANTE MPS 801

La MPS 801 permette di stampare sia caratteri alfanumerici e grafici presenti nella tastiera del VIC-20, sia disegni punto per punto. La sua utilizzazione è tipicamente perciò quella di presentare i risultati di certe elaborazioni oppure i listati dei programmi.
L'apertura di un canale di comunicazione con la stampante avviene tramite la solita istruzione OPEN; la MPS 801 ha la possibilità di variare

il numero di dispositivo tramite un interruttore posto nel retro: è possibile perciò selezionare sia il numero di dispositivo 4 che il 5 e perciò sono collegabili al VIC-20 due MPS 801 dato che possono avere un numero di dispositivo distinto.
Il comando necessario per listare un programma sulla stampante, predisposta come dispositivo #4, è molto semplice:

```
OPEN 4, 4 : CMD 4 : LIST
```

a cui deve seguire, una volta terminata la stampa, il comando, in modo diretto, di chiusura del canale:

```
PRINT# 4: CLOSE 4
```

È possibile selezionare due insiemi di caratteri nella stampa: quello maiuscolo-grafico (è in pratica il set di caratteri attivato all'atto dell'accensione del VIC), e quello minuscolo-maiuscolo. La selezione dei due set di caratteri avviene durante l'apertura del canale di comunicazione con la stampante utilizzando due diversi indirizzi secondari:

OPEN 4 , 4 , 0 caratteri maiuscoli-grafici
OPEN 4 , 4 , 7 caratteri minuscoli-maiuscoli

Oltre a questo sono possibili diversi modi di stampa selezionabili mediante l'invio di un opportuno carattere di controllo, CHR$(x), nelle istruzioni PRINT#, CMD, OPEN. La tabella 3.4 prospetta le varie possibilità.

Tabella 3.4

Carattere di controllo	*Descrizione*
chr$(8)	modo grafico
chr$(10)	avanzamento della carta
chr$(13)	ritorno carrello
chr$(14)	dimensione doppia dei caratteri
chr$(15)	dimensione standard
chr$(16)	posizionamento della testa di scrittura
chr$(17)	caratteri minuscoli-maiuscoli
chr$(18)	caratteri in negativo
chr$(26)	ripetizione del carattere grafico
chr$(27)	specifica l'indirizzo del punto (grafico)
chr$(145)	caratteri maiuscoli-grafici
chr$(146)	caratteri in positivo

Di tutti i vari modi di stampa quello che presenta una certa difficoltà di programmazione è il grafico; in questo modo è possibile far stampare un singolo punto, forse è meglio chiamarlo pixel, nella posizione che si vuole nel foglio di carta.
È una opzione che assomiglia molto, come gestione, alla realizzazione di grafici in alta risoluzione nello schermo televisivo: anche per la stampante occorre definire un carattere grafico e fornirle la forma utilizzando delle istruzioni DATA.
L'esempio che segue, ricavato dal manuale della 801, dà una procedura per costruire un carattere grafico non presente nel set di caratteri della stampante.
Si vuole far stampare il carattere:

```
. . * * . . .
. * . . * * .
* . . . * . .
* . . . . . .
* . . . * . .
. * . . * * .
. . * * . . .
```

ove gli asterischi rappresentano un punto di stampa.
Occorre riportare su un foglio di carta la forma del carattere e numerarne le righe nel modo seguente:

```
1    . . * * . . .
2    . * . . * * .
4    * . . . * . .
8    * . . . . . .
16   * . . . * . .
32   . * . . * * .
64   . . * * . . .
```

e per ognuna delle sette colonne sommare i numeri delle righe in cui appare l'asterisco e aggiungere 128. Si ottengono allora i valori:

156 162 193 193 182 162 0.

Questi sette valori debbono essere utilizzati in una istruzione DATA per passare l'informazione alla stampante. Il programma che segue mostra

un tipico esempio della procedura da adottare:

```
 10 DATA 156, 162, 193, 193, 182, 162, 0
 20 FOR I = 1 TO 7
 30 READ A
 40 A$ = A$ + CHR$(A)
 50 NEXT
 60 OPEN 4 , 4
 64 :
 65 REM MODO GRAFICO
 70 PRINT# 4, CHR$(8) A$
 74 :
 75 REM MODO NORMALE
 80 PRINT# 4, CHR$(15)
 90 CLOSE 4
100 END
```

Associato al modo grafico, anche se può benissimo essere utilizzato nel modo normale, è il comando per predisporre il punto di inizio della stampa. Il comando ha la seguente struttura:

```
PRINT #4, CHR$(16) CHR$(X1) CHR$(X2);
```

dove CHR$(16) è il comando di predisposizione del punto di inizio della stampa, mentre CHR$(X1) e CHR$(X2) individuano una delle 80 colonne nell'ambito di una riga in cui deve iniziare la stampa. Per lo stesso scopo si può usare più agevolmente:

```
PRINT #4, CHR$(16)"xx";
```

dove xx è il numero di colonna da cui inizia la stampa.
Ad esempio se si vuole far iniziare la stampa a partire dalla colonna 25-esima occorre dare l'istruzione:

```
PRINT #4, CHR$(16)CHR$(50)CHR$(53);
```

(si noti che 50 e 53 sono i valori ASCII rispettivamente della cifra 2 e della cifra 5) o, molto più semplicemente:

```
PRINT# 4, CHR$(16)"25";
```

Per quanto riguarda la descrizione degli altri comandi si rimanda il lettore al manuale della 801.
Al fine di mettere in evidenza le caratteristiche grafiche della stampante

nella figura 3.4 è riportato un disegno ottenuto tramite il programma di grafica DOODLE.

Fig. 3.4 Figura eseguita con il DOODLE.

IL PRINTER PLOTTER 1520

Il 1520 della Commodore è sia una stampante a bassa velocità sia un dispositivo di tracciamento di grafici. Lo scambio di informazioni con il calcolatore avviene tramite il bus seriale 488, come del resto succedeva anche con la MPS 801.
Come stampante esso può scrivere con quattro diverse dimensioni di carattere: 10, 20, 40, 80 caratteri per riga, la riga essendo lunga 96 millimetri. La scrittura può avvenire in quattro colori diversi: nero, rosso, verde e blu, selezionabili con opportuni comandi.
Per quanto riguarda le sue capacità di plottaggio esso è in grado di disegnare su un'area di 96 × 400 millimetri, anche in questo caso con i quattro diversi colori.
I comandi da inviare al 1520 per la selezione delle diverse modalità di funzionamento si differenziano solo per l'indirizzo secondario, come è indicato nella tabella 3.5.
Per ogni tipo di operazione si deve aprire un canale, con numero di dispositivo pari a 6, e indirizzo secondario scelto a seconda dell'operazione voluta, e inviare, tramite un PRINT# il parametro opportuno.

Tabella 3.5

Operazione	*Indirizzo secondario*
stampa di caratteri	0
plottaggio	1
selezione colore	2
selezione dimensione	3
selezione rotazione	4
selezione tratteggio	5
selezione maiuscolo-grafico / minuscolo-maiuscolo	6
reset del plotter	7

Per selezionare le varie operazioni si può utilizzare la seguente serie di comandi OPEN:

```
OPEN 4, 6
OPEN 1, 6, 1
OPEN 2, 6, 2
OPEN 3, 6, 3
OPEN 9, 6, 4
OPEN 5, 6, 5
OPEN 6, 6, 6
OPEN 7, 6, 7
```

OPEN 4, 6 permette di far stampare i caratteri inviati dal calcolatore nel normale set maiuscolo grafico. Se il 1520 è stato appena acceso i caratteri sono 40 per riga, il colore è nero.
OPEN 1, 6, 1 apre il canale di comunicazione per il plotter; esistono 6 comandi in questo modo di operazione, comandi che debbono essere inviati al 1520 tramite una istruzione PRINT# opportuna:

```
PRINT# 1, "comando" (, X, Y)
```

dove X e Y sono le eventuali coordinate associate ai comandi descritti di seguito.

H	selezione del punto di origine delle coordinate assolute
I	selezione del punto di origine delle coordinate relative
M	posizionamento della penna nel punto di coordinate assolute X Y

D	traccia dalla posizione attuale fino a quella di coordinate assolute X Y
R	posizionamento della penna nel punto di coordinate relative X Y
J	traccia dalla posizione attuale fino a quella di coordinate relative X Y

Di seguito sono elencati esempi dei sei tipi di comandi per il plotter:

```
OPEN 1, 6, 1
PRINT# 1, "M", 50, 1
PRINT# 1, "D", 300, −30
PRINT# 1, "I"
PRINT# 1, "R", 30, 340
PRINT# 1, "J", 2, −15
PRINT# 1, "H"
```

OPEN 2, 6, 2 permette di selezionare il colore della penna tramite l'istruzione:

```
PRINT# 2, C
```

ove C vale:

Valore di C	*Colore*
0	nero
1	blu
2	verde
3	rosso

ad esempio per far scrivere o plottare in colore verde:

```
OPEN 2, 6, 2: PRINT#2, 2
```

OPEN 3, 6, 3 permette invece di scegliere la grandezza dei caratteri; ciò si ottiene con:

```
PRINT# 3, G
```

ove G vale:

Valore di G	*Dimensione*
0	80 caratteri/linea
1	40 caratteri/linea
2	20 caratteri/linea
3	10 caratteri/linea

L'indirizzo secondario 4 permette di scegliere la rotazione dei caratteri, i quali possono essere stampati normalmente o ruotati di 90 gradi a destra. I comandi sono:

```
OPEN 9, 6, 4 : PRINT# 9, 0
```

per l'orientazione normale, e

```
OPEN 9, 6, 4 : PRINT# 9, 1
```

per la rotazione.
Per la selezione nel modo di stampa o di plottaggio a tratteggio occorre inviare un comando con indirizzo secondario 5 e fornire un parametro, con valore tra 0 e 15 compresi, per imporre le caratteristiche di tratteggio. Il parametro di valore 0 non dà tratteggio, quello con valore 15 dà il tratteggio con i tratti distanziati al massimo. Il comando da inviare per il tratteggio è del tipo:

```
OPEN 5, 6, 5 : PRINT# 5, T
```

La selezione dei due set di caratteri disponibili è attuabile con l'invio di un comando con indirizzo secondario 6 e con un parametro, di valore 0 oppure 1, che impone rispettivamente il normale set maiuscolo-grafico o quello minuscolo-maiuscolo.
Anche in questo caso il comando da dare è molto semplice:

```
OPEN 6, 6, 6: PRINT# 6, S
```

Infine il comando di reset del 1520 alle condizioni iniziali, cioè quelle in cui si trova appena viene acceso, si attua molto semplicemente con:

```
OPEN 7, 6, 7: PRINT# 7 : CLOSE 7
```

Nella figura 3.5 è illustrato un disegno effettuato con il plotter.

FIGURE GEOMETRICHE

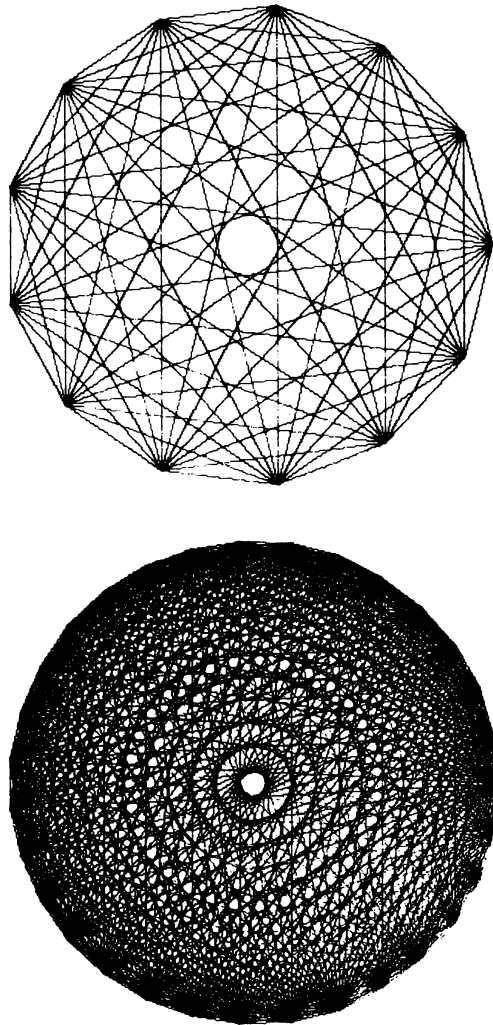



Fig. 3.5 Figure realizzate con un plotter.

LA PORTA SERIALE RS232-C

Nel VIC-20 è disponibile una porta seriale, secondo lo standard RS 232-C, per lo scambio di informazioni con quei dispositivi, tipicamente stampanti e modem, che utilizzano questo protocollo di comunicazione.

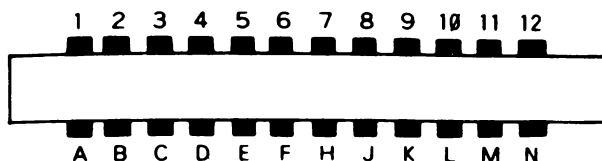
La porta seriale è formata, nella versione più semplice, da tre linee: una di trasmissione, una di ricezione e una di riferimento per le tensioni. I dati sono trasmessi in forma di sequenze di impulsi: un singolo byte diventa quindi una sequenza di otto impulsi. I valori di tensione previsti dallo standard sono di ± 12 volt: è necessario allora collegare al VIC-20 una opportuna interfaccia per normalizzare i segnali a tale valore di tensione.
Nel VIC-20 è prevista anche la gestione della porta seriale con segnali di *handshaking*, cioè con un certo numero di linee dedicate alle segnalazioni fra i due dispositivi che stanno colloquiando, di modo che quello che deve trasmettere dei dati lo fa solo se dal ricevente gli arriva una segnalazione di "pronto a ricevere"; allo stesso modo il trasmittente invia al ricevente la segnalazione di essere pronto a trasmettere.
Questo modo di comunicazione è tipico per il collegamento con il modem, dispositivo che permette la trasmissione di informazioni attraverso una linea telefonica e quindi trasforma i livelli di tensione in segnali fonici. In ricezione esso effettua la conversione da segnali fonici a livelli logici.
Il connettore presente nel VIC-20 è descritto in figura 3.6, mentre il connettore secondo lo standard RS232-C è visibile in figura 3.7.
La gestione della porta seriale è effettuata in modo automatico dal VIC-20: esso si incarica di "serializzare" i dati in trasmissione e di trasformarli in byte in ricezione, tutto ciò in base alle specifiche di

PIN	TYPE	NOTE	PIN	TYPE	RS232 FUNCTION
1	GND		A	GND	
2	+5V	100mA MAX	B	CB1	
3	RESET		C	PB0	SIN
4	JOY0		D	PB1	— RTS
5	JOY1		E	PB2	— DTR
6	JOY2		F	PB3	— RI
7	LIGHT PEN		H	PB4	— DCD
8	CASSETTE SWITCH		J	PB5	
9	SERIAL ATN IN		K	PB6	— CTS
10	+9V	100mA MAX	L	PB7	— DSR
11	GND		M	CB2	— SOUT
12	GND		N	GND	— GND

Fig. 3.6 Il connettore RS232 del VIC.

PIN		
1	Protective Ground	AA
2	Transmitted Data	BA
3	Received Data	BB
4	Request To Send	CA
5	Clear To Send	CB
6	Data Set Ready	CC
7	Signal Ground	AB
8	Carrier Detect	CF
9	(not used)	
10	"	
11	"	
12	"	
13	"	
14	"	
15	"	
16	"	
17	"	
18	"	
19	"	
20	Data Terminal Ready	CD
21	(not used)	
22	"	
23	"	
24	"	
25	"	

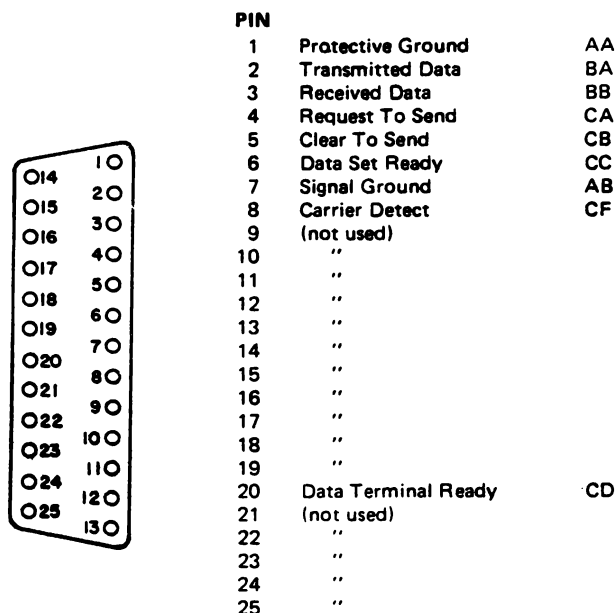



Fig. 3.7 Il connettore RS232 standard.

comunicazione fornite dall'operatore e ricavate da quest'ultimo in base alle caratteristiche dell'apparecchiatura collegata.
Durante una sessione di comunicazione il calcolatore riserva un'area di memoria di 512 byte nella parte alta della memoria RAM disponibile e ciò allo scopo di realizzare due buffer, uno di trasmissione e uno di ricezione, da 256 byte ciascuno. È da notare che in un programma Basic l'apertura di un canale per la RS232-C deve avvenire prima di qualsiasi operazione su variabili di tipo stringa per evitare la perdita di tali variabili nel momento in cui il KERNAL realizza i due buffer.

Registri dedicati alla RS232-C

Il sistema operativo del calcolatore utilizza tre registri per implementare la RS232: essi sono il registro di stato, quello di controllo e quello di comando.
Il contenuto degli ultimi due è automaticamente predisposto, come si vedrà, nel momento in cui si apre il canale per la rice-trasmissione seriale.
Il registro di stato è invece continuamente aggiornato in modo automatico da parte del sistema operativo in modo che l'operatore possa

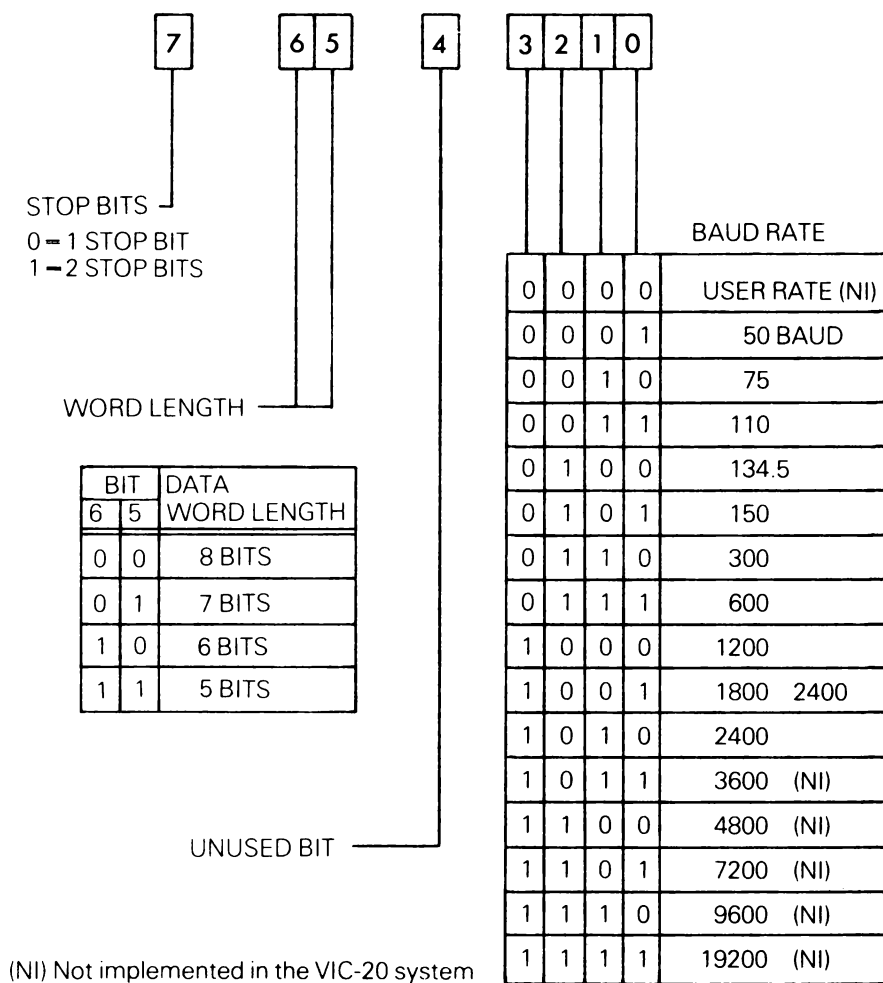



Fig. 3.8 Funzione dei bit nel registro di controllo del VIC.

immediatamente rilevare eventuali problemi che insorgessero durante una sessione di ricetrasmissione.

Il registro di controllo

Questo registro è usato per imporre la velocità di trasferimento dei dati e per specificare da quanti bit sono composti. Queste due informazioni debbono essere desunte dal manuale della periferica collegata tramite la RS232.
La velocità di scambio dei dati deve essere fornita in bit al secondo, o

baud: se la velocità è di 600 baud, e ciascun carattere è trasmesso con otto bit più uno di stop e uno di parità, quest'ultimo per il controllo della correttezza del dato in ricezione, allora la velocità di trasmissione è di circa 60 caratteri al secondo.
I primi quattro bit meno significativi del registro di controllo impongono la velocità in baud. I tre più significativi impongono invece il numero di bit di stop e il numero di bit con cui è codificato un carattere da trasmettere (o da ricevere); in figura 3.8 sono descritte le varie possibili

BIT 7	BIT 6	BIT 5	OPERATIONS
		0	Parity disabled, none Generated/Received
0	0	1	Odd Parity Receiver/Transmitter
0	1	1	Even Parity Receiver/Transmitter
1	0	1	Mark Transmitted Parity Check Disabled
1	1	1	Space Transmitted Parity Check Disabled

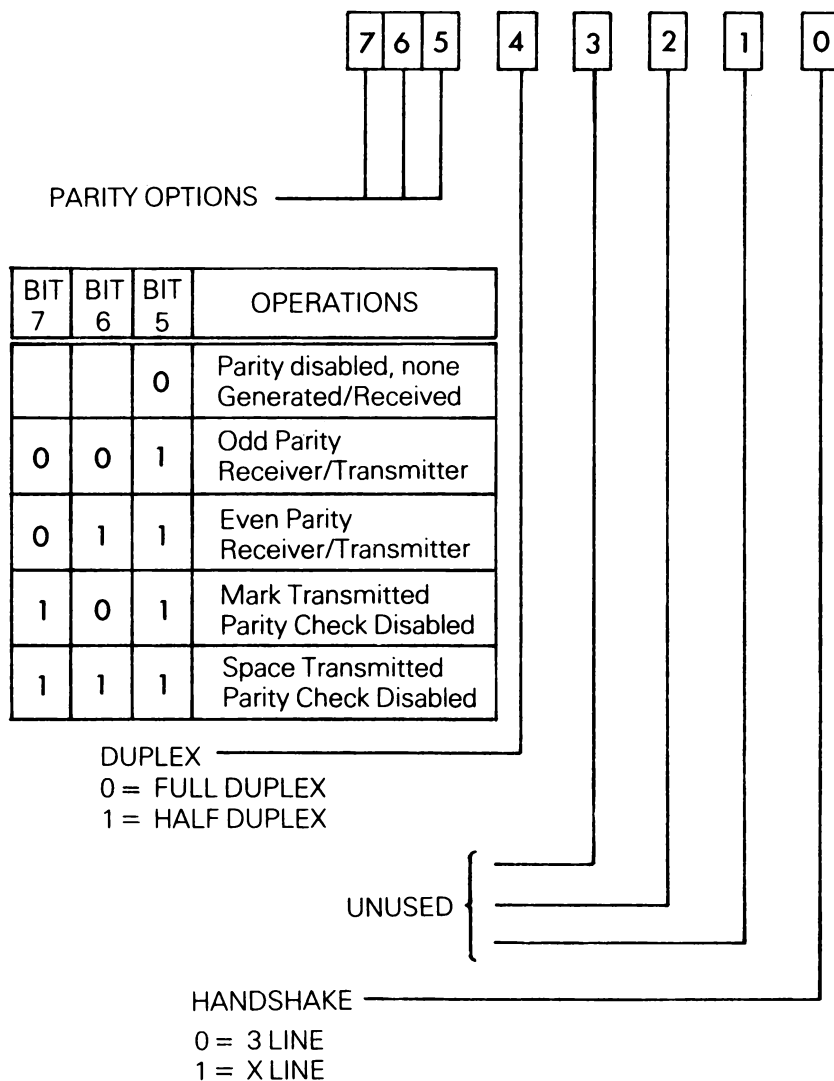



Fig. 3.9 Funzione dei bit nel registro di comando del VIC.

scelte: ad esempio se si vuole trasmettere a 2400 baud, con 2 bit di stop e una lunghezza di carattere di 7 bit, il contenuto del registro di controllo deve essere pari a 170.

Il registro di comando

Esso controlla il modo con cui è effettuata la rice-trasmissione; i suoi bit hanno le funzioni indicate in figura 3.9.
Ad esempio, se si vuole attuare una ricetrasmissione a 3 linee, in half duplex e parità pari, occorre che il registro di controllo contenga il valore 113.

Il registro di stato

Il KERNAL aggiorna continuamente il registro di stato per la RS232 durante tutto il tempo in cui è aperto un canale per la trasmissione seriale.

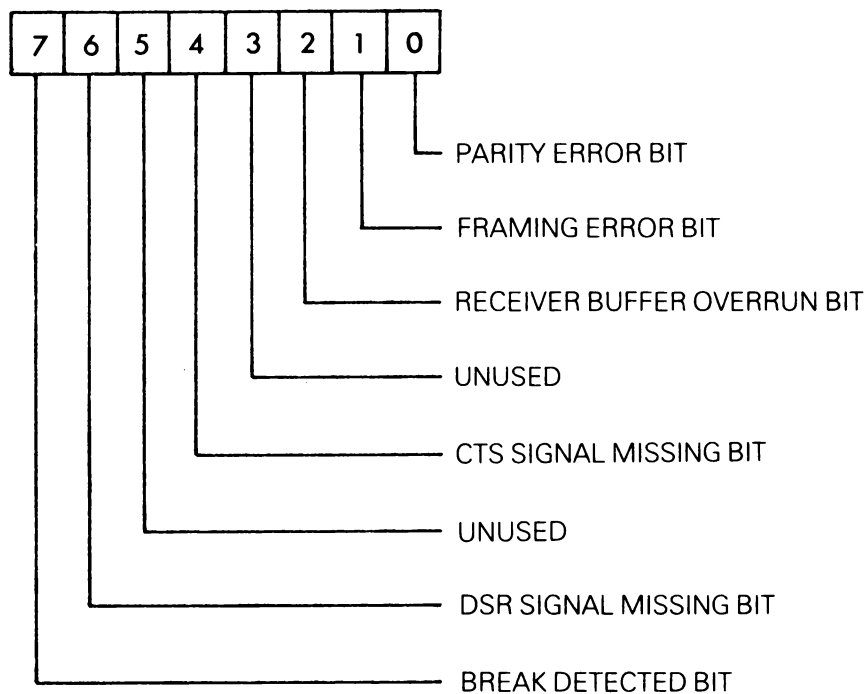



Fig. 3.10 Funzione dei bit nel registro di stato del VIC.

Il significato dei vari bit di questo registro è descritto in figura 3.10. Per la RS232-C a tre linee i bit che interessano sono il meno significativo, quello di ordine 2 e il più significativo.
La lettura del contenuto del registro di stato è necessaria prima di effettuare la chiusura del canale dedicato alla RS232 per evitare la perdita di dati ricevuti e non ancora utilizzati (o quella di dati non ancora trasmessi) dato che la chiusura del canale rende non più disponibili i contenuti dei buffer di trasmissione e di ricezione.

APERTURA DI UN CANALE RS232

L'istruzione per aprire il canale seriale (il sistema operativo può gestire un solo canale di questo tipo alla volta), è:

```
OPEN FL , 2 , 0 , CHR$(CT) + CHR$(CM)
```

ove FL è il numero di canale e 2 individua la RS232; i CHR$(CT) e CHR$(CM) sono i caratteri ASCII equivalenti ai valori CT e CM imposti ai registri di ConTrollo e di CoMando rispettivamente.
Per aprire il canale secondo quanto detto nei due esempi precedenti basta dare la:

```
OPEN FL, 2, 0, CHR$(170) + CHR$(113)
```

RICEZIONE DI DATI DA RS232

A tale scopo basta far eseguire la:

```
GET #FL, A$
```

la quale permette di acquisire, uno alla volta, i caratteri presenti nel buffer di ricezione seriale. Se il buffer è vuoto, A$ è la stringa nulla; si noti inoltre che se i dati ricevuti sono formati da meno di otto bit, allora vengono messi automaticamente degli zeri nei corrispondenti bit mancanti del byte A$.

TRASMISSIONE DI DATI NELLA RS232

I comandi Basic disponibili in questo caso sono due:

```
CMD LF
```

e

```
PRINT #2, dato
```

si noti che i dati da trasmettere sono prima inviati al buffer di trasmissione e poi trasmessi in modo automatico contemporaneamente all'esecuzione di un programma Basic.

CHIUSURA DEL CANALE RS232

Il comando da dare è:

```
CLOSE FL
```

È da tenere presente che la chiusura del canale provoca la perdita dei caratteri eventualmente presenti nel buffer di ricezione poiché il KERNAL rende l'area occupata dai buffer disponibile per la memorizzazione delle variabili di tipo stringa.
Allo scopo di evitare inoltre la mancata trasmissione di qualche dato conviene allora testare la variabile ST e il bit 6 della locazione 37151 con le due istruzioni:

```
nn  IF ST=0 AND (PEEK(37151) AND 64 )=1 THEN nn
kk  CLOSE FL
```

I JOYSTICK

Il connettore "Control port" che si trova sul lato destro del VIC-20 è destinato al collegamento di tre dispositivi diversi: il joystick, le paddle e la light-pen. In questo paragrafo vedremo il software necessario per gestire i joystick.

Joystick

Il joystick è collegato in modo che i suoi cinque interruttori, quelli che si chiudono quando lo si sposta nella direzione nord, sud, est e ovest, oltre a quello di Fire, sono gestiti dalle due VIA.
Una di queste è però utilizzata dal KERNAL per la gestione della tastiera e quindi per la lettura dei valori dei joystick occorre disabilitare tale modo di funzionamento, per ripristinarlo quando si aspettano dati o comandi dalla tastiera.
Il modo più semplice per effettuare una lettura del joystick è quello dato qui di seguito:

```
POKE 37154, 127
X%= ( NOT (PEEK(37151)) AND 60 - ((PEEK(37152) AND 128) =0 )
POKE 37154, 255
```

Le tre istruzioni in pratica fanno sì che nella variabile X% siano contenute tutte le informazioni relative allo stato dei joystick, dato che a ogni interruttore dello stesso è associato un bit ben preciso della variabile X%.
È perciò agevole verificare lo stato dei cinque interruttori secondo quanto indicato di seguito:

```
FIRE   : X% AND 32
EST    : X% AND 16
SUD    : X% AND  8
NORD   : X% AND  4
OVEST  : X% AND  1
```

In questo modo risulta facile sapere se il joystick è spostato in una certa direzione: ad esempio se si vuole far eseguire una certa routine se il joystick punta a nord basta l'istruzione:

```
IF ( X% AND 4 ) THEN...
```

Per poter avere dal joystick una informazione completa del suo stato, cioè una indicazione della sua posizione, conviene definire due variabili: una che dà una indicazione di NORD-SUD e una di EST-OVEST con le istruzioni:

```
NS = SGN (X% AND 1) – SGN (X% AND 16)
EO = SGN (X% AND 8) – SGN (X% AND 4)
```

Le variabili NS ed EO assumono i valori 0 se non c'è spostamento in quella direzione, 1 verso la direzione data dalla prima lettera della variabile, −1 verso quella data dalla seconda lettera.
Ad esempio se risulta NS = −1 ed EO = −1 ciò vuol dire che il joystick al momento della lettura era verso SUD-OVEST.

I file su registratore

Il VIC-20 permette di memorizzare su nastro magnetico (e su disco) due tipi di file (= archivio, elenco, schedario) che si differenziano a seconda che riguardino programmi o dati.
Qualunque sia il tipo di file esso è comunque sempre memorizzato come una sequenza di byte: ciò che diversifica i due tipi è il modo con cui essi sono gestiti dal calcolatore e dall'operatore.
I file programma sono in pratica la copia delle locazioni della RAM del calcolatore dove è contenuto il programma Basic e sono gestite in modo automatico dal sistema operativo tramite le istruzioni **SAVE** e **LOAD**.
I file di dati invece contengono liste di variabili numeriche o di tipo stringa che l'operatore ha voluto salvare su nastro per un utilizzo successivo; possono essere costituiti da elenchi di oggetti, risultati di certe elaborazioni, insomma da qualsiasi cosa l'utilizzatore voglia archiviare. La gestione dei file di dati è lasciata completamente alla responsabilità dell'operatore dato che egli solo sa quello che vuole registrare, l'ordine con cui registrare e il numero di dati. I comandi necessari alla creazione e alla lettura di file di dati sono: **OPEN, CLOSE, INPUT#, GET#, PRINT#.**
Per la gestione del registratore il sistema operativo riserva un'area di memoria RAM detta *tape buffer*, che va dall'indirizzo **828**, ‹**033C**›, al **1019**, ‹**03FB**›, per un totale di 192 locazioni.
Tutti gli scambi di informazioni tra il VIC e il registratore avvengono sempre tramite il buffer e quindi in blocchi di 192 byte.

FILE DI PROGRAMMA

Quando si salva un programma sul nastro, il sistema operativo crea nel buffer un primo blocco, detto *header*, che contiene:

1. il byte meno significativo dell'indirizzo di inizio della RAM di programma (cioè quello contenuto in locazione 43);
2. il byte più significativo del medesimo indirizzo (locazione 44);
3. il byte meno significativo dell'indirizzo in cui si trova l'ultimo carattere del programma (locazione 45);
4. il byte più significativo del medesimo indirizzo (locazione 46).

Per un VIC senza espansioni di RAM aggiunte i primi due byte hanno i valori 1 e 16 rispettivamente: infatti $1 + 16 \times 256 = 4097$, che è proprio l'indirizzo dello *start of Basic*; gli ultimi due byte hanno invece dei valori che dipendono ovviamente dalla lunghezza del programma.
A questi due byte seguono eventualmente quelli relativi ai caratteri alfanumerici che costituiscono il nome con cui si vuole registrare il programma (al massimo 16 caratteri). Sono poi inviati tanti byte di valore 32 fino al riempimento del buffer.
Il sistema operativo, effettua quindi la trasmissione del contenuto del buffer verso il registratore. Successivamente sono inviati al registratore i primi 192 byte relativi al programma vero e proprio, quindi ancora altri 192 byte e così via fino a che il KERNAL si accorge di avere trasmesso tutti i byte del programma.
A questo punto il KERNAL ripete la trasmissione dell'header, del primo blocco, del secondo e così via.
Questa seconda trasmissione è effettuata allo scopo di verificare, al momento di una lettura del programma da nastro, la correttezza dei byte letti, ed eventualmente di correggere quelli che risultassero affetti da errori.
Il controllo degli errori è possibile dato che per ogni byte di programma il KERNAL trasmette in realtà un gruppo di 9 bit: 8 relativi al byte effettivamente da trasmettere e uno, di parità, calcolato in modo che la somma dei bit di valore 1 presenti nei 9 bit sia un numero pari.
Al momento della lettura del nastro, il KERNAL verifica che per ogni gruppo di 9 bit sia soddisfatta la regola della parità; se ciò non accade esso memorizza in RAM, nelle locazioni da 256 a 318, l'indirizzo della locazione nell'area RAM di programma dove ha memorizzato il byte sospetto. Durante la lettura successiva dello stesso blocco di byte, il sistema andrà a verificare se i byte corrispondenti a quelli errati soddisfi-

no alla parità: se ciò accade il nuovo byte è sostituito a quello errato nella memoria di programma.
Se anche qualcuno di questi byte non soddisfa la parità, il KERNAL invia al video il messaggio:

LOADING ERROR

e cessa il caricamento del programma.
Si noti che un eventuale errore multiplo nello stesso byte può far sì che la parità sia soddisfatta: però nella seconda lettura di quel byte il KERNAL effettua sempre il paragone con il risultato della prima lettura e se li trova diversi dà ancora il messaggio di errore.
Questo metodo di rilevazione degli errori è abbastanza sicuro: e ciò è provato dalla poca frequenza con cui si ha il messaggio di errore. Quando non si riesce a caricare un programma ciò è dovuto in genere o all'uso di nastri di cattiva qualità o al fatto che le testine del registratore sono sporche.

FILE DI DATI

Nel caso si debbano registrare su nastro file di dati, il modo di operare del KERNAL è circa lo stesso. Diversamente dal caso dei file di programma ora il numero di dati da registrare non è conosciuto a priori dal KERNAL: allora la procedura di registrazione avviene in modo tale che ogni blocco è registrato due volte consecutivamente. Inoltre nell'header i primi quattro byte contengono l'indirizzo di inizio e di fine del tape buffer.
Per creare un file di dati su nastro si può usare il comando:

OPEN 1, 1, 1, "nome del file"

Questo provoca l'assegnazione del canale #1 al registratore (1), con indirizzo secondario 1, cioè per la scrittura. (L'indirizzo secondario 2 provoca la scrittura del file con al termine l'*End Of Tape,* EOT, cioè una particolare segnalazione di fine nastro).
Per scrivere sul nastro il valore della variabile N, o la stringa A$, si usa l'istruzione:

PRINT#1, N

oppure:

```
PRINT# 1, A$
```

In entrambi i casi il valore della variabile N o la stringa A$ sono memorizzati sul nastro e seguiti dal byte di ‹return› (= $0D).
Invece nel caso della istruzione:

```
PRINT# 1, N, A$
```

nel nastro sono memorizzati il valore N, seguito da undici caratteri di spazio, poi la stringa A$ seguita da $0D.
Se si usa l'istruzione:

```
PRINT#1, N; A$
```

nel nastro, ai caratteri corrispondenti al valore di N, seguono immediatamente quelli di A$.
In pratica nel nastro la memorizzazione avviene con le stesse modalità di un PRINT sullo schermo.
Si scriva, ad esempio, il seguente programma:

```
10 OPEN 1, 1, 1, "PROVA"
20 A=23: A$= "PROVA"
30 PRINT# 1, A: PRINT#1, A$
40 CLOSE 1: END
```

e lo si mandi in esecuzione: esso crea su nastro un file di dati che contiene proprio i valori 23 e la stringa "PROVA".
Per quanto riguarda la lettura di un file di dati occorre aprire un canale di lettura da registratore tramite, ad esempio:

```
OPEN 1, 1, 0, "nome programma"
```

e fare eseguire istruzioni

```
INPUT # 1, A
```

per leggere dati numerici, oppure

```
INPUT# 1, A$
```

se si tratta di stringhe.
Ad esempio se si vogliono leggere i dati scritti dal programma precedente basta far eseguire il programma:

```
10 OPEN 1, 1, 0, "PROVA"
20 INPUT #1, A, A$
30 PRINT A, A$
40 CLOSE 1: END
```

si vedrà scrivere sullo schermo il valore 23 e la stringa PROVA.
La lettura da nastro può anche avvenire un byte alla volta, utilizzando l'istruzione

```
GET# 1, C$
```

che è molto simile alla GET relativa alla tastiera.
Il programma:

```
10 OPEN #1, 1, 0, "PROVA"
20 FOR I = 1 TO 10
30 GET #1, C$: PRINT C$
40 NEXT I: END
```

ci permette di vedere come sono memorizzati nel nastro i byte relativi alla variabile N e alla stringa A$.
Il programma seguente contiene le procedure per creare un file di dati e per rileggerlo.

```
10 REM**************
11 REM*FILE DI DATI*
12 REM*SU NASTRO   *
13 REM**************
14 REM
20 PRINT"*****MENU'*****
30 PRINT
40 PRINT"  PER CREARE UN FILE 1 "
50 PRINT"  PER LEGGERE        2 "
60 PRINT"  PER FINIRE         3 "
70 GETA$: A=VAL(A$): IFA <1 OR A>3 THEN70
80 ON A GOTO1000,2000,3000
1000 PRINT"CREAZIONE DI FILE"
1010 PRINT
1020 PRINT"PREMI ↑ PER SALVARE IL TESTO SU NASTRO"
1030 PRINT
1040 INPUT " NOME DEL FILE";N$
1050 REM**************
1051 REM*APERTURA DEL*
1052 REM*   CANALE   *
1053 REM**************
1054 REM
1060 OPEN 1, 1, 1, N$
1070 PRINT" SCRIVI IL TESTO"
1080 INPUT T$
```

```
1090 IFT$="↑"THEN 1200
1100 PRINT#1,T$
1110 GOTO1080
1200 PRINT"  CHIUSURA DEL FILE"N$
1210 PRINT#1,"FINE":CLOSE1:GOTO10
2000 REM**************
2001 REM*LETTURA DI   *
2002 REM*  UN FILE    *
2003 REM**************
2004 REM
2010 PRINT"  QUALE FILE  VUOI RILEGGERE ?"
2020 INPUT N$
2030 REM**************
2031 REM*APERTURA DEL*
2032 REM*CANALE       *
2033 REM**************
2034 REM
2040 OPEN 1,1,0,N$
2050 INPUT#1, T$
2060 PRINT T$
2070 REM**************
2071 REM*CHECK STATUS*
2072 REM**************
2073 REM
2080 IF ST=0 THEN 2050
2090 REM**************
2091 REM* E' STATO   *
2092 REM*RILEVATO UN *
2093 REM*   ERRORE   *
2100 REM**************
2110 REM*OPPURE L'EOF*
2111 REM**************
2112 REM
2120 CLOSE 1
2130 GOTO 10
3000 END
```

Associato alla gestione di file è anche il comando:

CMD FL, "NOME"

che è usato assieme alla:

OPEN FL, ND

per far sì che il dispositivo di uscita dei messaggi dal KERNAL non sia più il video ma la periferica ND.
Ad esempio, l'istruzione scritta in modo immediato:

OPEN 1, 1: CMD 1,"LIST SU NASTRO": LIST

fa sì che venga formato su nastro il file dati LIST SU NASTRO, il quale contiene proprio il listing del programma come sarebbe apparso sul video, quindi comprensivo anche del messaggio finale READY.
Per far sì che il dispositivo di uscita torni ad essere il video, si deve dare il comando (sempre in modo immediato):

```
PRINT #1 : CLOSE 1
```

È da notare che la lista memorizzata sul nastro è formata dai byte corrispondenti ai caratteri che apparirebbero sul video, e non da quelli tokenizzati relativi al programma Basic residente in memoria RAM: perciò non è possibile caricare questo file come se fosse un programma e mandarlo in RUN.
Come si vedrà più avanti l'uso del comando CMD è indispensabile per effettuare il listing del programma sulla stampante.
Il KERNAL tiene conto dei possibili errori che avvengono durante una lettura da nastro settando opportunamente una variabile riservata, ST, secondo quanto indicato nella tabella 4.1

Tale variabile di stato è memorizzata in locazione 144 $0090; come si può notare a seconda del valore che essa assume sono indicate varie condizioni che possono essersi verificate durante la lettura del nastro.
La ST è utilizzata anche per tutte le comunicazioni che avvengono con dispositivi collegati alla porta seriale IEEE (dischi, stampante ecc.).
Per quanto riguarda il registratore, in tabella sono indicate due colonne: la prima a sinistra è relativa alla gestione di file di dati, la seconda ai programmi.

Tabella 4.1

Valore di ST lettura/scrittura	*Lettura da registratore*	*LOAD/VERIFY da registratore*	*Bus IEEE*
1 in write	–	–	time out
2 in read	–	–	time out
4	blocco corto	blocco corto	–
8	blocco lungo	blocco lungo	–
16	errore di lettura	errore di confronto	–
32	errore di checksum	errore di checksum	–
64	End Of File	–	EOF
128	End Of Tape	EOT	Dispositivo non presente

Per l'utilizzatore sono da verificare solo le condizioni:

16 = errore generico
32 = errore di checksum
64 = lettura del carattere di fine di file (EOF)
128 = lettura del carattere di fine nastro (EOT).

Conviene quindi, ogni volta che si leggono da nastro dei file sequenziali, effettuare la verifica della ST per assicurarsi che tutto proceda correttamente e in caso contrario per non rischiare di perdere quanto acquisito fino al momento dell'errore.
Il checksum (ST = 32) di cui sopra è un controllo della validità della lettura effettuato automaticamente dal KERNAL per assicurarsi di non aver preso per buoni due valori letti male ma che soddisfano le condizioni di uguaglianza e di corretta parità.

FILE DI BYTE

È possibile creare file sequenziali di byte e rileggerli un byte alla volta.
È il caso, ad esempio, in cui si vogliono salvare su nastro o su disco un'area di memoria, sia ROM che RAM, o semplicemente dei dati rappresentabili con un solo byte.
Per far ciò occorre utilizzare per la scrittura sul file la solita

PRINT#

e per la lettura la

GET# A$

che, come è noto, preleva un solo byte alla volta e lo assegna alla variabile A$.
L'esempio di programma di p. 83 memorizza sul nastro l'area di memoria che va dall'indirizzo SA fino all'indirizzo EA. È da sottolineare l'uso che si fa delle istruzioni CHR$(e ASC(: dato che sul nastro sono memorizzati solo dei byte in sequenza, di valore da 0 a 255, durante la scrittura del file occorre trasformare il contenuto di ogni locazione di memoria in un carattere alfanumerico, che, come sappiamo, è identificato dal VIC proprio con un byte.
Per far ciò se C è il valore letto nella memoria, la

C$ = CHR$(C)

genera proprio il carattere, o meglio il byte, corrispondente al valore di C. Si noti che tale carattere può non essere visualizzabile sullo schermo video; ad esempio la successione di valori numerici:

86, 73, 67, 2, 0

è trasformata dalla CHR$(nella successione di caratteri:

V, I, C,??,??

dove i doppi punti interrogativi indicano che non si tratta di caratteri visualizzabili sullo schermo.
Occorre inoltre tenere presente che la registrazione, avvenendo per byte, fa sì che nel nastro siano memorizzate delle stringhe alfanumeriche; come tali esistono delle limitazioni al momento della lettura (un byte di valore zero non è un carattere di una stringa ma solo il delimitatore della stringa stessa; il massimo numero di caratteri di una stringa è limitato ecc.).
Al momento della lettura del nastro l'istruzione:

```
GET# 1, A$
```

preleva una stringa formata da un solo carattere, quello individuato dal byte letto. Se tale byte vale 0, la stringa letta è nulla ("") e risulta poco agevole utilizzarla così com'è per ricavarne il valore numerico associato. (Si provi a dare il comando: PRINT ASC(""): il calcolatore dà errore!.)
È per questo motivo che durante la lettura di un file di byte occorre usare le istruzioni:

```
GET# 1, C$: A = ASC(C$ + CHR$(0))
```

Per far sì che se C$ = "", ad A sia proprio assegnato il valore 0.
Il programma che effettua una memorizzazione dell'area di memoria che va dall'indirizzo iniziale SA al finale EA su nastro è:

```
10 PRINT CHR$(147)
20 INPUT "INDIRIZZO INIZIALE";SA
30 INPUT "INDIRIZZO FINALE"; EA
40 INPUT"NOME DEL FILE";N$
50 OPEN1,1,1,N$
60 PRINT#1,SA:PRINT#1,EA
70 FOR I = SA TO EA
```

```
 80 PRINT#1,CHR$(PEEK(I));
 90 NEXT
100 CLOSE1:END
```

Tale programma può essere utilizzato come subroutine, per memorizzare ad esempio un certo numero di dati provenienti dal convertitore analogico/digitale descritto nel capitolo 8.
Il programma che segue può invece essere utilizzato per rileggere il file sequenziale creato dal programma precedente:

```
10 PRINT CHR$(147)
20 INPUT "NOME DEL FILE";N$
30 OPEN1,1,0,N$
40 INPUT#1,SA:INPUT#1,EA
50 FOR I=SA TO EA:
60 GET#1,A$:A = ASC(A$+CHR$(0))
70 POKE I,A
80 NEXT
90 CLOSE1:END
```

Il Video Interface Chip 6561

Il VIC è dedicato completamente alla generazione dell'informazione video e del suono. Esso contiene tutta l'elettronica necessaria per fornire l'informazione riguardante la forma e il colore dei caratteri presentati sullo schermo televisivo.
Il 6561 è inoltre in grado di generare effetti sonori e contiene due convertitori analogico-digitali.
È possibile programmare il 6561 in modo che esso esegua certi compiti ottemperando alle istruzioni fornite dall'utente. Questo è necessario quando si vogliano utilizzare nuovi caratteri oppure quando si voglia effettuare una gestione grafica dello schermo televisivo.
Prima di addentrarci in queste due tematiche conviene vedere, anche se in modo superficiale, come si attua la programmazione del VIC.
I 16 registri interni al 6561 sono accessibili con indirizzi a partire da 36864; scrivendo opportune configurazioni di byte in questi registri è possibile programmare il circuito in modo che faccia quello che vogliamo. La scrittura su tali registri può essere realizzata con istruzioni POKE, in Basic, o con le corrispondenti istruzioni in linguaggio macchina.
Di seguito sono brevemente descritte le funzioni dei registri di programmazione.

Registro #1 (36864 $9000)

I bit 6-0 determinano la posizione dell'informazione video fornita dal calcolatore rispetto al margine sinistro dello schermo televisivo. Il bit 7 è inutilizzato. Il programma che segue permette di osservare il movimento orizzontale dello schermo generato dal calcolatore:

```
10 FOR I = 0 TO 50
20 POKE 36864, I
30 FOR T = 1 TO 500: NEXT
40 NEXT I
50 POKE 36864, 12 : END
```

Registro #2 (36865 $9001)

Predispone la posizione dell'informazione video, come il registro #1, ma riferita al margine superiore dello schermo televisivo, come si può vedere facilmente facendo eseguire il programma:

```
10 FOR I = 0 TO 100
20 POKE 36865, I
30 FOR T = 1 TO 500 : NEXT
40 NEXT I
50 POKE 36865, 38 : END
```

Registro #3 (36866 $9002)

I bit 6-0 determinano il numero di colonne con cui è presentata la informazione video. Il bit 7 contribuisce a imporre l'indirizzo della memoria RAM video e di quella che contiene l'informazione relativa al colore dei caratteri. Per rendersi conto di come varia l'immagine dello schermo al variare del contenuto del registro #3 basta far eseguire il programma:

```
10 FOR I = 128 TO 255
20 POKE 36866, I
30 FOR T = 1 TO 500 : NEXT
40 NEXT I
50 POKE 36866 , 150
```

Registro #4 (36867 $9003)

In questo registro i bit 6-1 impongono il numero di righe con cui è presentata l'informazione video. Il bit 0 permette di selezionare caratteri composti da 64 da 128 bit. Il bit 7 è associato al contenuto del registro #5 per identificare il numero di linea di scansione del pennello elettronico nel televisore. Il programma mostra le conseguenze dovute a modifiche del contenuto di tale registro:

```
10 FOR I = 128 TO 178
```

```
20 POKE 36867, I
30 FOR T = 1 TO 500 : NEXT
40 NEXT I
50 POKE 35867, 174 : END
```

Registro #5 (36868 $9004)

Questo registro contiene il numero di linea attualmente scandita dal pennello televisivo.

Registro #6 (36869 $9005)

L'utilizzo di questo registro sarà ampiamente descritto in seguito; per ora basti dire che il suo contenuto individua sia l'indirizzo di inizio della matrice dei caratteri che quello dalla RAM video.

Registro #7 (36870 $9006)

Il contenuto di tale registro individua la posizione orizzontale di una eventuale penna luminosa collegata al calcolatore.

Registro #8 (36871 $9007)

Individua la posizione verticale della penna luminosa.

Registro #9 (36872 $9008)

Contiene il valore digitalizzato relativo a uno dei due potenziometri collegabili al connettore per le paddle.

Registro #10 (36873 $9009)

Contiene il valore digitalizzato relativo al secondo potenziometro di cui sopra.

Registro #11 (36874 $900A)

Permette di predisporre la frequenza di funzionamento del primo oscillatore, o generatore di nota. Il bit 7 deve valere 1 perché sia generata la nota.

Registro #12 (36875 $900B)

È analogo al registro #11 ma riguarda il secondo oscillatore.

Registro #13 (36876 $900C)

È come il #11 ma riguarda il terzo oscillatore.

Registro #14 (36877 $900D)

Permette di predisporre la frequenza del generatore di rumore con modalità identiche a quelle dei tre registri precedenti.

Registro #15 (36878 $900E)

I bit 3-0 permettono di predisporre il volume del suono generato dai quattro oscillatori precedenti. I bit 7-4 permettono di scegliere il colore ausiliario nel modo di visualizzazione multicolore.

Registro #16 (36879 $900F)

I bit 7-0 di questo registro permettono di scegliere uno tra i 16 colori possibili per lo sfondo dello schermo televisivo. I bit 2-0 permettono di scegliere uno tra gli 8 colori per il bordo dello schermo. Il bit 3 permette di scegliere se i caratteri o i simboli visualizzati sullo schermo sono di colori differenti ma con lo stesso colore di fondo oppure sono di colore uguale ma su differenti colori di sfondo.

Il registro #6

Questo è il registro più importante del 6561 e forse anche quello di più difficile gestione. In esso il bit 7 deve avere sempre il valore logico 1; i bit 3-0 individuano l'indirizzo di inizio della matrice di caratteri utilizzata, mentre i bit 6-4, assieme al bit 7 del registro 36866, individuano l'inizio della memoria (RAM!) di schermo. Incominciamo a vedere come è individuata quest'ultima: i bit 6-4 del registro 36869 vanno a formare i bit 12-10 del bus degli indirizzi quando il 6561 vuole leggere la memoria di schermo: quindi, in base al valore di questi tre bit la locazione di inizio della stessa può essere solo una di quelle di indirizzo:

Tabella 5.1

0	0	0	1	0	0	4096
0	0	0	1	0	1	5120
0	0	0	1	1	0	6144
0	0	0	1	1	1	7680

Questi tre bit sono però usati insieme a un bit del registro 36866, il settimo, il quale forma il bit 9 del bus degli indirizzi: allora le possibili locazioni di inizio della RAM di schermo sono:

Tabella 5.2

0	0	0	1	0	0	0	4096
0	0	0	1	0	0	1	4608
0	0	0	1	0	1	0	5120
0	0	0	1	0	1	1	5632
0	0	0	1	1	0	0	6144
0	0	0	1	1	0	1	6656
0	0	0	1	1	1	0	7168
0	0	0	1	1	1	1	7680

. . .
36869
36866

Se si suppone che la matrice dei caratteri rimanga fissa all'indirizzo 32768, i valori che debbono essere posti nei registri 36866 e 36869 per imporre una certa area di memoria di schermo, sono indicati di seguito:

Tabella 5.3

Area schermo	36866	36869	648
4096	22	192	16
4608	150	192	18
5120	22	208	20
5632	150	208	22
6144	22	224	24
6656	150	224	26
7168	22	240	28
7680	150	240	30

Per quanto riguarda i bit 3-0, che contribuiscono, come già detto, a individuare l'inizio della RAM/ROM ove è posta la matrice dei caratteri, diciamo subito che il bit 3 indica che si tratta di RAM o di ROM a seconda che esso valga 1 o 0 rispettivamente; i bit 2-0 vanno a costituire nell'ordine i bit 12-10 dell'indirizzo di inizio. Ciò è sintetizzato nella tabella 5.4.

Tabella 5.4

3 2 1 0	inizio
0 0 0 0	32768 ROM
0 0 0 1	33792 ROM
0 0 1 0	34816 ROM
0 0 1 1	36864 ROM
1 1 0 0	4096 RAM
1 1 0 1	5120 RAM
1 1 1 0	6144 RAM
1 1 1 1	7168 RAM

La tabella che segue indica invece i valori che si debbono scrivere nella locazione 36869 per allocare l'inizio dell'area contenente la matrice dei caratteri, nell'ipotesi che l'area di schermo inizi a 7680:

Tabella 5.5

Inizio matrice	36869	3 2 1 0
4096	252	1 1 0 0
5120	253	1 1 0 1
6144	254	1 1 1 0
7168	255	1 1 1 1

Utilizzando le informazioni contenute nelle due tabelle precedenti è relativamente facile trovare i valori da mettere nel registro 36869 per allocare dove vogliamo sia l'area di schermo che quella di caratteri. Ad esempio si vuole che la prima inizi a partire dalla locazione 4096 e la seconda a partire dalla 5120.
Dalla tabella 5.5 si ricava:

```
bit 7 6 5 4 3 2 1 0
    ? ? ? ? 1 1 0 1        (= 13)
      (memoria caratteri inizia a 5120)
```

il bit 7 deve essere a 1 per indicare che si tratta di RAM:

```
bit 7 6 5 4 3 2 1 0
    1 ? ? ? 1 1 0 1        (= 141)
```

Dalla tabella 5.2 si ottengono i valori dei tre bit rimanenti:

```
bit 7 6 5 4 3 2 1 0
    1 1 0 0 1 1 0 1      (= 205)
```

occorre anche che il bit 7 del registro 36866 sia a 0.
Allora per dare questa informazione al VIC occorre fornire il comando:

```
POKE 36869,205: POKE 36866,22
```

Si noti che occorre fornire anche al sistema operativo l'informazione di dove si trova l'area di schermo. Questo si può ottenere con l'istruzione:

```
POKE 648,16
```

dato che la locazione 648 deve contenere il numero di pagina dove inizia l'area di schermo.
Prima di descrivere i metodi con cui si possono definire nuovi caratteri o effettuare della grafica conviene riassumere brevemente le modalità con cui il VIC-20 genera l'informazione video:

a. ogni carattere presente sullo schermo video è formato da un insieme di otto byte e quindi da 8 × 8 = 64 bit; ogni bit corrisponde a una posizione ben definita nell'ambito del carattere; se un bit vale 1, sullo schermo si "accende" un puntino (o pixel = minima area controllabile) del carattere.
b. il normale schermo generato dal VIC è composto da 23 righe di 22 caratteri ciascuna, cioè sono definite 23 × 22 = 506 posizioni distinte per i caratteri.
c. dato che ogni carattere è di 8 × 8 pixel lo schermo ha 22 × 8 = 176 pixel in ogni riga di scansione e 23 × 8 = 184 righe di scansione: ciò porta ad un totale di 176 × 184 pixel.

Il dispositivo che gestisce l'informazione da inviare allo schermo televisivo è, come abbiamo visto, il 6561: sfruttando la possibilità di programmarlo è abbastanza agevole sia definire nuovi caratteri video sia realizzare della grafica.

DEFINIZIONE DI NUOVI CARATTERI

Come è noto il 6561, per generare l'informazione da inviare allo schermo televisivo, preleva i codici di carattere presenti nelle 506 locazioni

della memoria di schermo. Il valore di ogni codice è moltiplicato per otto e sommato all'indirizzo dove inizia, in ROM (O RAM) l'area di memoria in cui risiede l'informazione relativa alla "forma" dei caratteri; l'indirizzo così ottenuto fornisce al 6561 la prima delle otto locazioni in cui è memorizzata l'informazione video relativa al carattere corrispondente a quel codice.
Ad esempio se in una certa posizione dello schermo è visualizzato il carattere "A" nella corrispondente locazione in memoria schermo è contenuto il byte di valore 01 (si ricordi che i codici di schermo non corrispondono ai codici ASCII); nella memoria di caratteri, o matrice di caratteri, che normalmente inizia in locazione 32768, a partire dalla locazione 32768 + 8 x 1 = 32776 fino alla 32783 sono presenti i byte di valore:

Indirizzo	*Valore*	*Byte*
32776	24	00011000
32777	36	00100100
32778	66	01000010
32779	126	01111110
32780	66	01000010
32781	66	01000010
32782	66	01000010
32783	0	00000000

Si può notare immediatamente che gli 1 presenti sono l'immagine di come appare la A sullo schermo televisivo.
È allora facilmente intuibile come si possano costruire e poi visualizzare nuovi caratteri: basterà informare il 6561 di dove si trova la nuova matrice di caratteri e in questa si dovranno scrivere gruppi di otto byte che nel loro insieme "formino" il carattere voluto; da quanto appena detto è evidente che la nuova matrice deve risiedere in RAM.
Se ad esempio si vuole che al posto del carattere @ sia visualizzato il nuovo carattere grafico "quadrato", poiché il codice della @ è 0, a partire dalla locazione di inizio in RAM della nuova matrice dei caratteri dovranno essere scritti gli otto byte:

Numero d'ordine	*Byte*	*Valore*
xxx0	11111111	255
xxx1	10000001	129
xxx2	10000001	129
xxx3	10000001	129
xxx4	10000001	129
xxx5	10000001	129

xxx6	10000001	129
xxx7	11111111	255

Occorre a questo punto definire la matrice dei caratteri la quale, come abbiamo visto, può risiedere in RAM a partire da una delle locazioni 4096, 5120, 6144 e 7168 e darne conto al 6561.
Si potrebbe pensare di sceglierne una a caso e fornire l'indirizzo corrispondente al 6561.
Bisogna però tenere presente che l'interprete Basic necessita di un'area di memoria continua in RAM in cui devono risiedere i programmi e le relative variabili: in un VIC-20 senza espansioni di memoria aggiunte tale area va dalla locazione 4096 alla 7679.
Se ci si limita nel numero di nuovi caratteri, si può allocare la matrice a partire dall'indirizzo 7168: facendo questo si hanno a disposizione 7680 − 7168 = 512 locazioni che ci permettono di definire 64 caratteri.
Se si vuole inoltre mantenere un minimo dei soliti caratteri alfanumerici, cioè quelli che hanno un codice di schermo da 0 a 57 compresi, è possibile definire solo 6 nuovi caratteri, quelli che hanno un codice di schermo di valore 58 - 63.
Per non dovere calcolare i byte dei caratteri che abbiamo mantenuto uguali ci conviene copiarli dalla "vecchia" matrice in ROM: il programma che segue ci permette di farlo e di definire, tramite le istruzioni DATA, la forma dei sei nuovi caratteri:

```
 1 REM*SPOSTAMENTO DELLA MATRICE DEI CARATTERI*
 2 :
10 POKE 36869, 255
11 :
12 REM*DIMINUZIONE DELLA MEMORIA DISPONIBILE PER IL
   BASIC*
13 :
20 POKE 56, 28 : POKE 52, 28: CLR
21 :
30 REM******************************
31 REM* TRASFERIMENTO DEI
32 REM* VECCHI CARATTERI
33 REM******************************
40 FOR I = 0 TO 58*8-1
50 POKE 7168 + I , PEEK(32768 + I)
60 NEXT
61 :
65 REM*SCRITTURA DEI NUOVI CARATTERI
70 FOR I = 58*8 TO I+6*8-1
80 READ C%
```

```
 90 POKE 7168 + 1, C%
100 NEXT
110 DATA x,x,x,x,x,x,x,x     : CARATTERE #1
120 DATA y,y,y,y,y,y,y,y     : CARATTERE #2
130 DATA z,z,z,z,z,z,z,z     : CARATTERE #3
140 DATA w,w,w,w,w,w,w,w : CARATTERE #4
150 DATA h,h,h,h,h,h,h,h     : CARATTERE #5
160 DATA k,k,k,k,k,k,k,k     : CARATTERE #6
```

Una volta mandato in esecuzione questo programma sono disponibili i sei nuovi caratteri associati ordinatamente ai codici di schermo **58-63** cioè ai tasti:

: ; < = > ?

In altre parole, se si vuole che sia visualizzato il carattere numero 4 occorre digitare **PRINT "="** oppure si può scrivere nella RAM di schermo il valore **61** tramite un **POKE**.

GRAFICA

Per realizzare della grafica le cose da effettuare sono concettualmente simili a quelle appena viste ma con una importante differenza: nelle locazioni dell'area di schermo sono presenti dei codici di caratteri tutti diversi tra loro e che non debbono mai essere modificati; invece la matrice dei caratteri deve essere aggiornata man mano che si realizza il disegno sullo schermo televisivo.
Per semplicità si immagini che lo schermo grafico sia di sei righe per sei colonne: le 36 posizioni corrispondenti nella memoria di schermo debbono contenere allora i codici indicati nella tabella:

Tabella 5.6

Numero di riga	*Numero di colonna*					
	0	1	2	3	4	5
0	0	1	2	3	4	5
1	6	7	8	9	10	11
2	12	13	14	15	16	17
3	18	19	20	21	22	23
4	24	25	26	27	28	29
5	30	31	32	33	34	35

Durante il processo di formazione dell'immagine video il 6561 va ancora a leggere tutte le 36 locazioni della memoria di schermo e i codici così ottenuti sono utilizzati per leggere nella matrice dei caratteri la "forma" dei caratteri stessi.
Ora è proprio nella matrice dei caratteri, che evidentemente deve risiedere su RAM, che debbono essere scritte le informazioni ed effettuate le operazioni necessarie per far visualizzare il disegno grafico voluto.

```
* * . . . . . .        (128 + 64)
. . * * . . . .        ( 32 + 16)
. . . . * * . .        (  8 +  4)
. . . . . . * *        (  2 +  1)
. . . . . . . *        (    1   )
. . . . . . . *        (    1   )
. . . . . . . *        (    1   )
. . . . . . . *        (    1   )
```

Fig. 5.1

Se ad esempio si vuole che sullo schermo video, nella zona corrispondente al carattere di codice video 1, cioè in corrispondenza della colonna 1 e riga 0, sia visualizzata la figura 5.1 occorre scrivere nella matrice di caratteri a partire dalla locazione iniziale incrementata di 8 (è il carattere di codice 1 !) la sequenza di valori:

192 48 12 3 1 1 1 1

Un altro esempio: si vuole tracciare la diagonale che inizia dall'angolo sinistro in alto dello schermo (dal carattere di codice 0) e termina nell'angolo in basso a destra (codice del carattere 35). Occorre in questo caso modificare la matrice dei caratteri in corrispondenza dei gruppi di otto locazioni associati ai codici video 0, 7, 14, 21, 28, 35. È allora necessario effettuare dei calcoli per individuare i 64 bit che debbono essere settati a 1 per "accendere" i relativi pixel sullo schermo.
È ovvio demandare questo compito al calcolatore: infatti si può individuare ogni pixel dello schermo ridotto con una ascissa X e una ordinata Y, ambedue di valore compreso tra 0 e 47 dato che ci sono 6×8 pixel in ogni riga e in ogni colonna.
Il pixel di coordinate X e Y pari a 0,0 coincide con l'angolo in alto a sinistra dello schermo, quello di coordinate 47,0 con quello in alto a destra e così via.
Se si vuole che sia acceso il pixel di coordinate, ad esempio, $X=11$ e $Y=2$, si devono individuare la riga e la colonna relative al carattere che

si trova in quella posizione e, nell'ambito degli otto byte che "formano" il carattere, quello in cui deve essere posto a 1 il bit corrispondente al pixel.
Dato che il pixel ha ascissa $X=11$ esso è contenuto in uno dei caratteri appartenenti alla colonna 1: infatti INT(11/8) = 1.
Occorre individuare anche la riga dove si trova il carattere: esso appartiene alla riga 0 dato che INT(2/8) = 0. Allora si deve modificare, nella matrice dei caratteri, quello che ha il codice di schermo pari a 1 dato che è il carattere presente nella riga 0 e nella colonna 1.
A questo punto occorre scoprire quale degli otto byte del carattere 1 contiene il bit corrispondente al pixel da accendere; gli otto byte possono essere numerati da 0 a 7 come indicato nella figura 5.2, dove è segnato il pixel di coordinate $X=11$ e $Y=2$:

	Numero d'ordine dei bit								
	0	1	2	3	4	5	6	7	
0		.		.		.	.	.	
1		.		.		.	.	.	
2		.		*	.	.	.	.	
3		.		.	.	.	.	.	carattere
4		.		.	.	.	.	.	numero 1
5		.		.	.	.	.	.	
6	.	.		.	.	.	.	.	
7	.	.	.	.	.	.	.	.	

Fig. 5.2

La riga, nell'ambito delle 8 possibili, si può ricavare dalla relazione:

R = Y AND 7 (dà come risultato 2)

e in modo analogo la colonna:

C = X AND 7 (dà come risultato 3)

Finalmente si hanno tutti gli elementi per accendere il pixel dato che ora si sa che:

a. appartiene al carattere che ha il codice di schermo pari a 1
b. è nel secondo byte di quel carattere
c. è il bit di valore 2 ↑ (7 − C) = 16

Perciò basta dare il comando:

POKE SM + 1 * 8 + 2, PEEK(SM + 1 * 8 + 2) AND 16

dove SM è l'indirizzo di inizio della nuova matrice di caratteri, per accendere il pixel di coordinate 11,2 nel nostro schermo ridotto. Quanto appena detto può essere realizzato dal programma seguente, il quale ci permette di far tracciare sullo schermo una qualsiasi curva di cui si conosca l'equazione.

```
 10 REM*****************
 20 REM*    GRAFICA    *
 30 REM*****************
 40 :
 50 REM*****************
 60 REM*  MATRICE DEI  *
 70 REM*   CARATTERI   *
 80 REM*    IN 5120    *
 90 REM*****************
100 POKE 36869,253
110 :
120 REM*****************
130 REM* PROTEZIONE    *
140 REM* DELLA MATRICE *
150 REM*****************
160 POKE 56,20:POKE 52,20:CLR
170 :
180 GOSUB 500
190 GOSUB 600
200 REM*****************
210 REM* PLOTTAGGIO    *
220 REM* DELLA CURVA   *
230 REM*****************
240 :
250 PI = 3.14
260 FOR X = 0 TO 175
270 Y = 43 + INT ( 42 * COS ( 4 * PI * X / 176 ))
280 GOSUB 700
290 GOSUB 900
300 NEXT
310 END
500 REM********************
510 REM*INSERZIONE DEI    *
520 REM*CODICI NELLA      *
530 REM*MEMORIA DI SCHERMO*
540 PRINT CHR$(147)
550 FOR I = 0 TO 241 :REM 11  RIGHE
560 POKE 7680 + I , I
570 POKE 38400 + I , 0 :REM COLORE NERO
580 NEXT : RETURN
590 :
600 REM*******************
610 REM*AZZERAMENTO DELLA*
```

```
620 REM*MEMORIA DFI        *
630 REM*CARATTERI          *
640 REM*******************
650 FOR I = 0 TO 241 * 8 + 7
660 POKE 5120 + I , 0
670 NEXT : RETURN
680 :
690 :
700 REM*******************
710 REM*  VERIFICA DI X   *
720 REM*******************
730 :
740 IF X < 0 OR X > 175 THEN STOP
750 :
760 :
800 REM*******************
810 REM*  VERIFICA DI Y   *
820 REM*******************
830 :
840 IF Y < 0 OR Y > 87 THEN STOP
850 RETURN
860 :
900 REM*******************
910 REM* ROUTINE GRAFICA *
920 REM*******************
930 :
940 R = INT ( Y / 8 ) : C = INT ( X / 8 )
950 M = 5120 + ( 22 * R + C ) * 8
960 RR = Y ANT 7 : CC = X ANT 7
970 POKE M + RR , ( 2 ↑ ( 7 - CC )) OR PEEK ( M + RR )
980 RETURN
```

CAPITOLO SEI

Il linguaggio macchina

Un programma in linguaggio macchina consiste in una sequenza di byte che forniscono nel loro insieme istruzioni e dati per il microprocessore.
In questa forma numerica il programma è difficile da scrivere ed è per questo motivo che a ogni istruzione di un microprocessore si associa un mnemonico che la individua in modo univoco: ad esempio al byte di istruzione di valore **$E8** è associato il mnemonico **INX** per il 6502.
Per scrivere allora un programma in linguaggio macchina si usano questi mnemonici (che sono diversi a seconda del microprocessore) e si utilizza un particolare programma, detto *assemblatore*, che si incarica di trasformarli nei byte corrispondenti. Esistono anche programmi *disassemblatori* che effettuano l'operazione inversa.
Le istruzioni in linguaggio macchina sono molto semplici nel senso che fanno eseguire al microprocessore azioni elementari: malgrado ciò un programma scritto in linguaggio macchina è eseguito con una velocità di un paio di ordini di grandezza superiore a quella con cui è eseguito un programma scritto in Basic. Inoltre un programma in linguaggio macchina permette di avere un controllo completo di tutte le risorse hardware della macchina.

I REGISTRI DEL 6502

Per poter scrivere un programma in linguaggio macchina occorre conoscere le caratteristiche del microprocessore; bisogna cioè sapere quali sono le risorse interne dello stesso. Per il 6502 esse sono riassunte in figura 6.1.

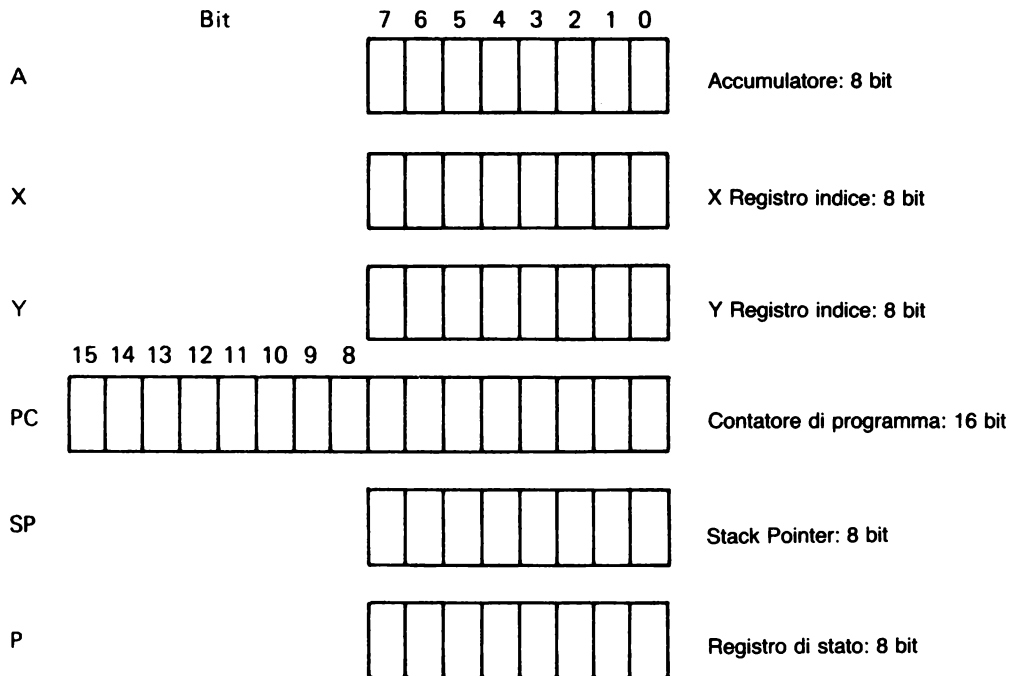



Fig. 6.1 Modello di programmazione del 6502.

Di seguito sono descritti i vari registri.

Accumulatore

È il registro più importante dato che è il solo utilizzato nelle operazioni aritmetiche o logiche.
È inoltre usato per trasferire dati da una locazione di memoria a un'altra. Ad esempio:

LDA #$ A0 carica nell'accumulatore il valore esadecimale A0
STA $ 1000 trasferisce il contenuto dell'accumulatore nella locazione 4096 decimale
ADC #$ 20 somma al contenuto dell'accumulatore il valore 32

Registro X

Anche questo registro può essere usato come l'accumulatore per depositarvi dei dati; le sue funzioni principali sono però quella di puntatore a locazioni di memoria e di contatore.

Registro Y

È simile al registro *X*, dal quale differisce per alcune funzioni.

Contatore di programma PC

Il PC è un registro da 16 bit che contiene l'indirizzo della prossima istruzione che il 6502 andrà ad eseguire: in questo modo il microprocessore esegue in sequenza tutte le istruzioni del programma dal principio alla fine.
Sono però sempre presenti in qualsiasi microprocessore delle istruzioni di salto condizionato che permettono di cambiare la sequenza di esecuzione delle istruzioni del programma all'insorgere di particolari condizioni.

Stack Pointer SP

Lo SP è usato soprattutto per la gestione di routine e di interruzioni: esso memorizza l'indirizzo di ritorno al programma principale.
Le routine sono gruppi di istruzioni, che eseguono un certo compito richiesto più volte durante l'esecuzione di un programma: invece di dovere inserire questi gruppi di istruzioni ogni volta che il programma principale ne ha bisogno, si attua una chiamata alla routine.
Essa, una volta eseguiti i suoi compiti, procede alla esecuzione della istruzione (del programma principale) immediatamente seguente a quella di chiamata. Questa istruzione è individuata proprio tramite lo SP.
Le interruzioni invece sono delle segnalazioni, in genere provenienti dall'esterno del calcolatore, che richiedono l'immediato arresto dell'esecuzione di un programma e l'esecuzione di particolari routine atte a gestire le interruzioni stesse.
Anche qui si ritorna a eseguire il programma interrotto una volta terminata l'esecuzione delle routine.

Registro di stato SR

Il registro SR memorizza nei suoi bit i valori di certi *flag* (segnalazioni, indicatori) che indicano particolari condizioni che si verificano durante l'esecuzione di un programma: è possibile allora far eseguire certi segmenti di programma piuttosto che altri facendo testare dal microprocessore i valori dei bit di flag.

I flag sono:

carry: indica se il risultato di una operazione aritmetica ha generato un riporto o un prestito; è aggiornato anche da istruzioni di paragone e da istruzioni logiche.
zero: esso è settato, cioè posto al valore logico 1, dopo ogni trasferimento di dati o dopo una operazione aritmetica se il contenuto del registro interessato a queste azioni è 0.
I: indica se il microprocessore è in grado di gestire o meno le interruzioni: se vale 0 queste possono essere gestite, se vale 1 no.
D: indica se le operazioni aritmetiche debbono essere effettuate in codice binario o decimale.
B: questo flag, di *break*, è settato o resettato dal 6502 per determinare durante una richiesta di interruzione se essa deriva dall'istruzione *break* o da un evento esterno.
O: questo flag, di trabocco, segnala se una operazione aritmetica ha modificato il valore del bit 7 dell'accumulatore.
N: flag di segno, indica se il risultato di una operazione aritmetica è negativo: in esso è trasferito il valore del bit 7 dell'accumulatore dopo ogni operazione aritmetica o logica.

È da sottolineare che il registro di stato è una parte essenziale di ogni microprocessore in quanto permette di prendere decisioni su quale deve essere la successiva parte di programma.

MODI DI INDIRIZZAMENTO

Le istruzioni del 6502 sono costituite da un codice operativo, espresso da un byte, seguito da 0, 1 o 2 byte di operandi.
Un byte permette di individuare 256 possibili istruzioni ma il 6502 utilizza solo 151 codici operativi. Nell'insieme di istruzioni del 6502 ci sono 56 istruzioni base ma sono possibili 13 diversi modi di indirizzamento che indicano su quale operando e/o su quale locazione di memoria l'istruzione agisce.
I modi di indirizzamento sono:

Immediato (2 byte): agisce direttamente utilizzando il secondo byte dell'istruzione stessa. Ad esempio:

```
LDX #$ AF
```

carica in X il valore $AF.

Assoluto (3 byte): in questo modo il secondo e il terzo byte dell'istruzione contengono l'indirizzo della locazione di memoria interessata dalla istruzione stessa. Ad esempio:

```
STA $ 1E00
```

memorizza il contenuto dell'accumulatore nella locazione $1E00.

Pagina zero (2 byte): il secondo byte dell'istruzione contiene l'indirizzo della locazione di memoria in pagina zero interessata dall'istruzione stessa. Ad esempio:

```
STX $30
```

memorizza il contenuto del registro X nella locazione $0030.

Implicito (1 byte): il registro interessato è implicito nella istruzione. Ad esempio:

```
SEI
```

setta il bit I nel registro di stato.

Relativo (2 byte): le istruzioni di salto condizionato contengono nel secondo byte l'indirizzo a cui effettuare il salto. Tale indirizzo è espresso come incremento o decremento rispetto all'indirizzo attuale. Ad esempio:

```
BEQ $ 1200
```

causa un salto all'istruzione contenuta nell'indirizzo $ 1200 se il flag Z è settato. L'istruzione è corretta solo se $1200 dista dalla locazione in cui risiede la BEQ di un valore compreso tra −126 e +129.

Indiretto (3 byte): si applica solo all'istruzione JMP; l'esecuzione del programma prosegue all'indirizzo contenuto nei due byte individuati dalla JMP. Ad esempio:

```
JMP($1300)
```

effettua il salto all'indirizzo contenuto in $ 1300 e in $ 1301.

Assoluto indicizzato (3 byte): in una istruzione che usa questo modo di indirizzamento l'indirizzo effettivo su cui agisce è dato dal contenuto del registro indice sommato all'indirizzo fornito dall'istruzione stessa.

Ad esempio:

STA $ 2000, X

memorizza il contenuto dell'accumulatore nell'indirizzo dato dal contenuto di X sommato a $ 2000.

Indicizzato in pagina zero (2 byte): è una forma dell'indicizzato che fa riferimento alla pagina zero. Ad esempio:

LDA $ A0, X

carica nell'accumulatore il contenuto della locazione di indirizzo $ A0 sommato al contenuto di X.

Indiretto pre-indicizzato (2 byte): in questo modo di indirizzamento l'indirizzo effettivo dell'operando è calcolato dal 6502 in questo modo: dapprima il valore contenuto nel registro X è sommato al valore dell'argomento; l'indirizzo risultante punta alla prima di due locazioni in pagina zero che contengono l'effettivo indirizzo dell'operando.
È quindi una combinazione dell'indirizzamento indiretto in pagina zero con l'indicizzazione. Ad esempio:

LDA ($ 29, X)

se X contiene il valore $07 viene caricato nell'accumulatore il contenuto della locazione di memoria individuata dai due byte presenti agli indirizzi $0029 e $0030.

Post-indicizzato indiretto (2 byte): può essere usato solo con il registro Y. Anche qui l'indirizzo effettivo è calcolato in due passi: il 6502 preleva dalla locazione indicata nell'istruzione un indirizzo a cui è sommato il valore contenuto nel registro Y; l'indirizzo risultante è quello effettivo dell'operando. Ad esempio, se Y contiene il valore $10:

LDA ($ 00), Y

se in $00 c'è il valore $20 e in $01, $AA, nell'accumulatore viene posto il valore contenuto in $AA30 ($AA20 + $10).
Un elenco delle istruzioni del 6502 è in appendice 2.

METODI DI SALVATAGGIO DI ROUTINE IN LINGUAGGIO MACCHINA

Per quanto riguarda il salvataggio di programmi in linguaggio macchina

occorre distinguere se si tratta di programmi a se stanti o di routine, di solito brevi, utilizzate nell'ambito di un programma Basic.
Nel primo caso i programmi sono scritti mediante assemblatori, più o meno completi, nell'ambito dei quali esistono sempre semplici comandi che permettono di salvare, su nastro o su disco, i programmi appena scritti e, soprattutto, reimmetterli successivamente nella memoria del calcolatore nelle stesse locazioni di memoria da cui erano stati salvati.
Accade molto spesso però che si scrivano corte routine in linguaggio macchina, le quali saranno utilizzate da un programma Basic, e che si vogliono caricare nella memoria del calcolatore contemporaneamente al programma stesso.
In questo capitolo saranno descritte alcune tecniche che ci permettono di risolvere agevolmente una necessità di questo tipo.
Per il salvataggio su nastro o su disco di routine in linguaggio macchina si possono utilizzare tre metodi: quello che fa uso di istruzioni DATA, quello che usa le REM e un terzo in cui si amplia l'area di memoria dedicata ai programmi in Basic.

Uso delle istruzioni DATA

Con le DATA in pratica si utilizzano istruzioni del Basic e quindi i byte del linguaggio macchina devono essere prima convertiti in valori decimali per essere poi allocati nelle posizioni di memoria specificate dal programmatore.
È inoltre compito di quest'ultimo proteggere l'area di memoria in cui risiederanno tali routine, in modo che il Basic non vi scriva sopra. A tale scopo occorre modificare, durante l'esecuzione del programma Basic, i puntatori 45 e 46, in modo che l'interprete Basic "veda" come occupate dal programma anche quelle locazioni che in realtà conterranno le routine in linguaggio macchina.
La struttura di un programma in Basic che utilizza questo metodo è del tipo:

```
 10 DATA .........
 20 DATA ..........
 30 ....
 40 ....
  .
  .
  .
100 FOR I = INIZIO TO FINE
110 READ A% : POKE I , A%
```

```
120 NEXT
130 :
140 REM PROTEZIONE DELLE ROUTINE
150 :
160 POKE 55 , INIZIO AND 255 : POKE 256 , INIZIO / 256 : CLR
170 :
 .
 .
200 REM PROGRAMMA BASIC
210 .....
220 .....
```

In pratica si riserva un'area di memoria alle routine in linguaggio macchina nella parte alta della memoria disponibile per il Basic, come si può osservare nel seguente schema:

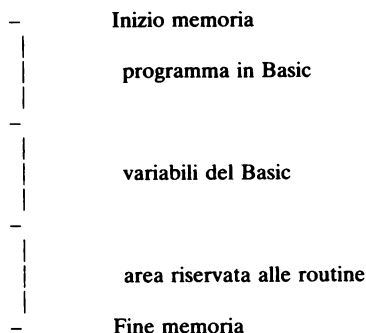



Fig. 6.2

Questo modo di procedere, anche se funziona correttamente, ha due difetti:

1. occorre effettuare la conversione dei codici relativi alle istruzioni in linguaggio macchina in valori decimali ‹0-255›.
2. Le istruzioni DATA occupano un'area di memoria molto più grande di quella necessaria per la memorizzazione dei codici in linguaggio macchina. A questa si deve aggiungere l'area dove effettivamente i codici di cui sopra saranno memorizzati (e protetti).

Il primo punto può facilmente essere ottimizzato nel senso che si può scrivere un programma in Basic che accetta in ingresso i codici delle istruzioni in linguaggio macchina, e cioè tipicamente in codice esadecimale, e li trasforma nei corrispondenti valori decimali.

Addirittura si può realizzare un programma che genera automaticamente istruzioni DATA, numerate a partire da 10000, ricavandone i valori mediante una lettura dell'area di memoria in cui precedentemente si era scritta la routine in linguaggio macchina:

```
49000 REM******************
49001 REM*RICHIESTA LIMITI*
49002 REM*DI INDIRIZZO PER*
49003 REM*LE ROUTINE IN L.M.
49004 REM******************
49005 :
50000 PRINT CHR$(147): INPUT "INDIRIZZO INIZIALE"; SA
50010 H% = SA / 256 : POKE 829 , H% : POKE 828 , SA - H% * 256
50020 INPUT "INDIRIZZO FINALE" ; EA
50021 :
50022 REM******************
50023 REMSALVATAGGIO DEI  *
50024 REM*LIMITI IN 828-832
50025 REM******************
50026 :
50030 H% = EA / 256 : POKE 831 , H% : POKE 830 , EA - H% * 256
50040 PRINT CHR$(147);
50041 REM******************
50042 REM* 10000 IN 833-834
50043 REM******************
50044 :
50050 POKE 833 , 234 : POKE 832 , 96
50051 :
50052 REM******************
50053 REM* NL E' IL NUMERO*
50054 REM* DELLA LINEA DATA
50055 REM* DA CREARE      *
50056 REM******************
50057 :
50060 NL = PEEK(833) * 256 + PEEK(832) : N$ = "000"
50061 :
50065 REM******************
50066 REM*AGGIORNAMENTO   *
50067 REM* INIZIO AREA    *
50068 REM******************
50069 :
50070 SA = PEEK(828) + PEEK(829) * 256:EA =PEEK(830) +
256*PEEK(831)
50074 REM******************
50075 REM* SIMULAZIONE DI *
50076 REM* INGRESSO DA    *
50077 REM*  TASTIERA
50078 REM******************
50079 :
50080 PRINT CHR$(147);NL;"DATA ";
50081 :
50085 REM******************
50086 REM*AGGIORNAMENTO  *
50087 REM*LINEA DATA     *
50088 REM******************
```

```
     50089
     50090 NL = NL + 10: HB% = NL / 256:POKE833,HB%:
POKE832,NL-HB%*256
     50091 :
     50095 REM******************
     50096 REM*LETTURA DI OTTO *
     50097 REM*BYTES DEL L.M.  *
     50098 REM******************
     50099 :
     50100 FOR NB = 0 TO 7 : BY = PEEK(SA + NB)
     50110 BY$ = RIGHT$( N$ + RIGHT$( STR$(BY),LEN(STR$(BY))-1),3)
     50120 IF NB < > 7 THEN BY$ = BY$ + ","
     50121 REM******************
     50122 REM*SCRITTURA DEL   *
     50123 REM* BYTE DI SEGUITO*
     50124 REM* NELLA DATA     *
     50125 REM******************
     50126 :
     50130 PRINT BY$ ;
     50140 NEXT NB
     50141 REM******************
     50142 REM*VERIFICA SE SONO*
     50143 REM*STATI LETTI TUTTI
     50144 REM* I BYTES        *
     50145 REM******************
     50146 :
     50150 IF SA + 8 > FA THEN 50210
     50160 PRINT:PRINT "RUN 50060"
     50170 SA = SA + 8 : HB% = SA / 256
     50180 POKE 829 , HB% : POKE 828 , SA - HB% * 256
     50190 POKE 631 , 19 : POKE 632 , 13: POKE 633,13: POKE198,3
     50200 END
     50210 POKE 631   19   POKE 632 , 13 : POKE 198 , 2
     50220 END
```

Le istruzioni dalla 50180 in poi meritano un commento un po' più esteso in quanto sono utilizzate per simulare, da programma, l'immissione di dati da tastiera.
Come è certamente noto al lettore le locazioni dalla 631 alla 640 costituiscono il buffer di tastiera nel quale sono successivamente deposti dal KERNAL i codici corrispondenti ai tasti via via attivati dall'operatore. La locazione 198 invece contiene il numero di codici presenti nel buffer di tastiera e non ancora prelevati o dal Basic o dal KERNAL.
L'istruzione 50190 mette nel buffer i codici corrispondenti a:

HOME; RETURN; RETURN

e pone nel contatore di caratteri presenti nel buffer il numero 3, per cui il calcolatore è avvisato che sono stati attivati, anche se non è vero, i tre tasti suddetti.

Durante un ciclo di lettura di otto byte si è scritta sullo schermo televisivo, nella prima riga, l'informazione seguente:

nnnnn DATA X1,X2,X3,X4,X5,X6,X7,X8

ove le Xi sono i valori decimali degli 8 byte letti ed nnnnn è il numero attuale NL dell'istruzione DATA che si vuole generare.
La 50160 fa sì che nella terza riga del video sia scritto:

RUN 50060

Quando si giunge a eseguire l'istruzione END, in 50200, il VIC-20 scrive come al solito READY ma poi si accorge che il buffer di tastiera contiene dei caratteri e li stampa sullo schermo dopo averli interpretati. Esegue perciò un HOME; con questo il cursore è proprio sulla riga che contiene nnnnn DATA ...., quindi un RETURN. Questo fa sì che il KERNAL creda che l'operatore abbia proprio immesso l'istruzione DATA e quindi la accetta come una nuova istruzione da aggiungere al programma Basic già presente in memoria.
Successivamente, per i medesimi motivi, esegue RUN 50060 che fa ripartire il programma di creazione delle istruzioni DATA.
Si noti che una volta terminata l'esecuzione del programma ci si trova ad avere nella memoria del calcolatore sia le istruzioni DATA appena generate sia il programma che ci ha permesso di farlo. Per l'inserzione delle DATA nel programma che effettivamente le utilizzerà e per l'eliminazione di quello che le ha generate si rimanda il lettore alla fine del prossimo paragrafo.
Riprendendo il nostro discorso, per quanto riguarda l'occupazione di memoria si consideri che:

a. ogni istruzione DATA è memorizzata dall'interprete Basic come una qualsiasi altra istruzione e quindi, oltre ai dati effettivi, sono impegnati due byte di link, due per il numero dell'istruzione, uno per il token DATA e uno per il byte di fine istruzione cioè un totale di 6 byte.
b. A ciò si deve aggiungere un numero di byte, variabile da uno a tre, per la memorizzazione dei valori decimali dei codici in linguaggio macchina, oltre al byte associato alla virgola.

Ad esempio, se si dovessero memorizzare 120 byte di linguaggio macchina, utilizzando 10 istruzioni DATA, si occuperebbe un'area di memoria variabile da un minimo di [$10 \times (6 + 2 \times 12) = 300$] a un massimo di [$10 \times (6 + 4 \times 12) = 540$] locazioni.
A queste si deve aggiungere un'area protetta di 120 locazioni ove risiederanno effettivamente le routine in linguaggio macchina, e quella

occupata dalle istruzioni Basic di lettura dei DATA e di riallocazione nelle locazioni di cui sopra.
È evidente l'enorme spreco di memoria che si ha procedendo in tal modo: ma lo si può evitare se si trascrivono direttamente nell'area assegnata le routine in linguaggio macchina, e si fa sì che tali routine siano contenute in un numero opportuno di istruzioni REM.

USO DELLE ISTRUZIONI REM

La procedura che si deve adottare è anche in questo caso abbastanza semplice: le prime istruzioni del programma in Basic sono delle REM seguite da un numero opportuno di simboli non usati nel programma Basic (ad esempio il simbolo £).
L'area di memoria occupata dalle routine in linguaggio macchina è letta un byte alla volta e questi byte sono sostituiti in sequenza nelle locazioni occupate dai caratteri £.
Evidentemente le routine in linguaggio macchina debbono essere forzatamente corte per poter essere contenute completamente in una sola istruzione REM: infatti dato che il numero di caratteri (= di byte) di una istruzione Basic è limitato a 88, non è possibile avere routine più lunghe di 87 byte (un byte è occupato dal token REM) e, se si verifica questa condizione, ciò impone noiosi calcoli per permettere il salto da una routine contenuta in una REM a una contenuta in un'altra.
Il programma che segue permette di allocare routine in linguaggio macchina in istruzioni REM; esso fornisce l'indirizzo di inizio della routine, cioè l'argomento della SYS, poi si debbono fornire al programma in successione i valori dei byte:

```
        10
REM£££££££££££££££££££££££££££££££££££££££££££££££
££££££££££££££££££££££££££££££££££££££££
        11
REM£££££££££££££££££££££££££££££££££££££££££££££££
££££££££££££££££££££££££££££££££££££££££
        12 .....
        13 .....
        .

        .
    20 REM**********************
    30 REM* PROGRAMMA **
    40 REM* BASIC     **
    50 REM**********************
```

```
  60 ...
  70 ...
  80 ...
  90 ...
 100 ...
 110 ...
 120 ...
 130 ...
 850 REM******************
 860 * MODIFICATORE *
 870 * DELLE REM*******
 890 REM******************
 895 :
 900 FOR I=4608 TO 4700
 910 IF PEEK(I)=ASC("£") THEN W=I : I=4700
 920 NEXTI
 930 PRINTCHR$(147); "!!!!SYS "; W; "!!!!"
 940 C=0
 950 INPUT"# OF BYTES"; N
 960 INPUT" CHECKSUM "; CK
 970 FORI=W TO W+N-1
 980 INPUT BY
 990 PRINTI, BY
1000 C=C+BY
1010 POKEI, BY
1020 NEXTI
1030 IF C <> CK THEN PRINT "CHECKSUM ERROR"
1040 END
```

Una volta mandato in esecuzione tale programma le istruzioni REM risultano modificate, il che appare evidente se si fa eseguire un LIST. A questo punto ci si trova ad avere oltre alle REM anche le istruzioni del programma che le ha generate e inoltre manca evidentemente il programma Basic che le deve utilizzare.

Allora la prima cosa da fare è eliminare le istruzioni del programma "generatore". Invece di editare i numeri di linea, ciò si può ottenere inserendo la seguente istruzione

```
1 PRINT"[illegible]"P:PRINT"P="P+1":L="L":IFP<=LTHEN1[illegible]":POKE198,2
:POKE631,13:POKE632,13:END

READY.
```

e dando poi il comando in modo immediato:

P = numero della prima istruzione da cancellare: L = numero dell'ultima istruzione: GOTO 1

Per quanto riguarda l'inserimento delle istruzioni REM così modificate in un programma Basic, si può utilizzare un qualsiasi metodo di *merge* (si veda il capitolo 2).

AMPLIAMENTO DELL'AREA BASIC

Un metodo migliore del precedente per quanto riguarda l'occupazione di memoria è quello di salvare assieme al programma l'area in codice macchina e di proteggerla durante l'esecuzione.
Si ottengono così i seguenti vantaggi:

1. l'occupazione di memoria è quella strettamente necessaria;
2. l'allocazione delle routine è effettuata solo una volta all'atto della scrittura delle routine stesse;
3. il procedimento è semplice e automatizzabile.

Con questo metodo si allocano le routine in linguaggio macchina subito dopo l'area Basic, avendo prima cura di variare i puntatori 45 e 46 in modo che il KERNAL veda come occupata da un programma Basic anche l'area di memoria che in realtà contiene il programma in linguaggio macchina.
È un metodo che risulta agevole se si possiede la cartuccia VIC-MON ma che è utilizzabile, con un po' più di difficoltà anche in caso contrario.
Conviene utilizzare la seguente procedura:

a. scrivere il programma Basic, verificarne il buon funzionamento senza però che siano mandate in esecuzione le istruzioni SYS... (ciò può essere ottenuto sostituendo le SYS con delle REM);
b. verificare dove si trova l'END OF BASIC e annotarne l'indirizzo;
c. con il VIC-MON inserire il programma in linguaggio macchina a partire dall'indirizzo di cui sopra;
d. verificare il buon funzionamento delle routine in linguaggio macchina ed effettuare le eventuali correzioni;
e. annotare l'indirizzo dell'ultima istruzione in linguaggio macchina e utilizzarne il valore per aggiornare i puntatori 45 e 46;
f. verificare che l'insieme dei programmi in Basic e in linguaggio mac-

china funzioni correttamente e se necessario ricominciare dal punto *a*;
g. infine salvare il programma con la solita

SAVE "nome programma",x ‹return›

In questo modo, nel momento di un successivo caricamento del programma, sono caricate sia le istruzioni Basic che quelle in linguaggio macchina e, soprattutto, queste ultime saranno protette riguardo a eventuali cancellazioni da parte del KERNAL.
Vediamo con un esempio come si può utilizzare la procedura appena descritta; il programma Basic abbia una struttura del tipo:

```
10 ..............
20 ..............
30 SYS.....
40 ..............
50 ..............
.
.
.
```

Il primo passo si attua scrivendo delle REM al posto delle SYS: con ciò la 30 si modifica in:

30 REM........

Si deve fare un po' di attenzione a non aggiungere simboli in più sulla riga 30 (il caso più frequente è l'inserzione di "spazio"), perché ciò modifica il valore dei puntatori 45 e 46.
Si dà il RUN al programma per verificarne il buon funzionamento e si effettuano le eventuali correzioni.
A questo punto si lancia il VIC-MON con la:

SYS 6 x 4096

e successivamente si dà il comando:

M 002D,002E

che ci permette di sapere l'indirizzo dell'END OF BASIC. Per ipotesi esso sia 13DA: allora l'assemblaggio delle routine in linguaggio macchina deve avvenire a partire da questo indirizzo, e lo si ottiene con:

.A 13DA xxxxxxx

eccetera.
Terminato l'assemblaggio si verificherà il buon funzionamento del programma in linguaggio macchina ed eventualmente si effettueranno le opportune correzioni.
Una volta che le cose funzionano occorre individuare l'indirizzo dove termina l'ultima istruzione in linguaggio macchina e aggiornare a tale valore i puntatori 45 e 46. Per ipotesi sia tale indirizzo $142C: allora si deve dare al VIC-MON il comando:

M 002D

e scrivere in $002D il valore $22 e in $002E il valore $14.
A questo punto si esce dal VIC-MON e si può salvare il programma dopo avere evidentemente ripristinato le istruzioni SYS.

CAPITOLO SETTE

Le routine del KERNAL

Si è già visto come si possano realizzare file sequenziali di dati e come con essi si possano memorizzare su nastro o su disco programmi scritti in linguaggio macchina proprio utilizzando tali tipi di file.
Il difetto insito in questo modo di procedere è che i file sequenziali possono essere riletti solo tramite istruzioni Basic, un byte alla volta che sarà poi immesso nella corretta locazione di memoria tramite una **POKE**. Ciò provoca una notevole perdita di tempo a causa della limitata velocità di esecuzione del Basic.
Perché allora non sfruttare le routine del KERNAL che effettuano **SAVE** e **LOAD** per memorizzare su nastro o su disco l'area di memoria che ci interessa come se fosse un'area di memoria di programma? Se si riesce a fare ciò si è in grado di caricare in memoria un qualsiasi insieme di istruzioni in linguaggio macchina in modo automatico senza dover passare tramite l'interprete Basic e dover subire la sua "lentezza".

LE ROUTINE

Nel KERNAL esistono quattro routine fondamentali che "presiedono" allo scambio di programmi tra il calcolatore e le memorie di massa esterne, registratore o disco; tali routine sono:

SETNAM: predispone il nome del file da caricare in memoria o da scrivere su nastro o disco.
SETLFS: predispone il canale, il numero di dispositivo interessato allo scambio e il suo indirizzo secondario.

SAVE: effettua il salvataggio di un'area di memoria sul dispositivo individuato da SETFLS col nome predisposto da SETNAM.
LOAD: carica un'area di memoria sempre in funzione di quanto predisposto da SETLFS e SETNAM.

Queste routine iniziano dalle locazioni:

$FFBA (65466)	SETNAM
$FFBD (65469)	SETLFS
$FED8 (65496)	SAVE
$FFD5 (65493)	LOAD

Occorre evidentemente fornire a tali routine i parametri necessari al loro funzionamento. Vediamo in dettaglio quali debbono essere questi parametri.

Routine SETNAM

A questa routine debbono essere forniti:

1. la lunghezza (cioè il numero di caratteri) del nome del file che si vuole trasferire: questo numero deve essere posto in locazione $030C (780);
2. l'indirizzo dove inizia la stringa di caratteri costituenti il nome; tale indirizzo deve essere immesso nelle locazioni $030D (781) e $030E (782) nel solito formato richiesto dal microprocessore 6502 e perciò in 782 la pagina in cui si trova la stringa, in 781 la locazione nell'ambito della pagina.

Si noti che le tre locazioni suddette corrispondono a quelle da cui il VIC-20 preleva il contenuto dell'accumulatore, del registro X e del registro Y, quando esegue una istruzione SYS..... (nel programma SAVE SU TAPE questo corrisponde alla riga:

ZK = PEEK(53) + 256 * PEEK(54) – LEN(T$) : POKE 782, ZK/256 : POKE 781,...: SYS 65469 ecc)

Routine SETLFS

Anche questa routine necessita di tre parametri che le debbono essere forniti rispettivamente nell'accumulatore, nel registro X e in quello Y: il primo è il numero di canale, il secondo il tipo di dispositivo (1 se si tratta del registratore, 8 se invece è il disco) il terzo l'indirizzo secondario.

Ad esempio se deve essere predisposto il canale #3 per il dispositivo 81 con indirizzo secondario 28 occorre dare le seguenti istruzioni in Basic:

```
POKE 780 ,  3
POKE 781 , 81
POKE 782 , 28
```

Routine di SAVE

La routine di SAVE deve essere mandata in esecuzione dopo che sono state eseguite SETLFS e SETNAM: occorre fornirle l'indirizzo di inizio e quello finale della zona di RAM da salvare.
L'indirizzo di inizio deve essere posto in pagina 0 in due locazioni successive che debbono contenere la parte meno significativa e quella più significativa dell'indirizzo stesso; inoltre deve essere fornito nell'accumulatore l'indirizzo della prima locazione di cui sopra.
L'indirizzo finale della RAM deve essere fornito invece nel registro X per quanto riguarda la parte meno significativa, nel registro Y per quella più significativa.
Ad esempio se S è la locazione di inizio, si può dare l'istruzione:

```
POKE 254 , S / 256
POKE 253 , S - PEEK(254) * 256
POKE 780 , 253
```

e se E è quella finale:

```
POKE 782 , E / 256
POKE 781 , E - PEEK(782) * 256
```

È da notare che per quanto riguarda il nastro la routine di SAVE non permette il salvataggio di locazioni superiori alla 32767 mentre per il disco non ci sono queste limitazioni.

Routine di LOAD

Tale routine può servire sia per effettuare un LOAD che per un VERIFY: ciò dipende dal contenuto dell'accumulatore che deve essere uguale a zero nel primo caso, a uno nell'altro. Al ritorno dalla routine nei registri X e Y è contenuto l'indirizzo in cui è stato memorizzato l'ultimo byte.
Il programma SAVE SU TAPE usa quanto appena detto per effettuare il salvataggio di una zona di RAM su nastro o su disco.

```
10 PRINTCHR$(14)
20 PRINT"■TTTINDIRIZZO INIZIALE":INPUTS
21 REM**************
22 REM*VERIFICA SE *
23 REM*L'INDIRIZZO *
24 REM* E' VALIDO  *
25 REM**************
26 :
30 IFS<256ORS>32767THENGOSUB290:GOTO20
40 PRINT:PRINT:PRINT:PRINT
41 :
42 :
50 PRINT"■TTTINDIRIZZO FINALE":INPUTE
51 REM**************
52 REM*VERIFICA SE *
53 REM*L'INDIRIZZO *
54 REM* E' VALIDO  *
55 REM**************
56 :
60 IFE<256ORE>32767THENGOSUB290:GOTO50
70 IFE<STHENPRINTC$;"■FINE   < INIZIO ":GOSUB220:GOTO50
71 :
72 :
73 :
80 PRINT:PRINT:PRINT
90 PRINT"]■---- SAVE ----■■■"
100 F$="":INPUT"■ NOME ";F$
110 PRINT:PRINT"■■■N■ASTRO O ■D■ISCO (N/D)"
115 REM**************
116 REM*  NASTRO  O *
117 REM*  DISCO  ?  *
118 REM**************
119 :
120 GETA$:IFA$<>"N"ANDA$<>"D"THEN120
130 DV=1-7*(A$="D"):IFDV=8THENF$="0:"+F$
131 :
132 :
134 REM**************
135 REM*IN X E Y    *
136 REM*L'INDIRIZZO *
137 REM*OVE E' IL   *
138 REM*  NOME      *
139 REM**************
140 T$=F$:ZK=PEEK(53)+256*PEEK(54)-LEN(T$):POKE782,ZK/256
150 POKE781,ZK-PEEK(782)*256:POKE780,LEN(T$):SYS65469:REM ** SETLFS **
151 :
160 POKE780,1:POKE781,DV:POKE782,1:SYS65466          REM ** SETNAM **
161 :
165 REM**************
166 REM*    SAVE    *
167 REM**************
169 REM:
170 POKE254,S/256:POKE253,S AND 255:POKE780,253
180 POKE782,E/256:POKE781,E AND 255:SYS65496
185 REM**************
186 REM* TEST DELLA  *
187 REM*   'ST '     *
188 REM**************
```

```
189 :
190 IF(PEEK(783)AND1)OR(ST AND191)THEN210
200 PRINT"FATTO.":END
210 PRINT"ERRORE DURANTE IL SAVE":PRINT"PROVA ANCORA":END
220 REM  BUZZER
230 POKE36878,15:POKE36874,190
240 FORW=1TO300:NEXTW
250 POKE36878,0:POKE36874,0:RETURN
260 :
270 :
290 PRINTC$;"NON PAGINA #0 O ROM":GOTO220

READY.
```

Esistono altre routine del KERNAL di uso generale e dedicate alla gestione di file; la loro utilizzazione permette di gestire in modo agevole specialmente lo scambio di informazioni con dispositivi collegati tramite il bus seriale IEE 488. Tali routine sono descritte brevemente qui di seguito:

Open logical file $FFC0

Questa routine, che deve essere preceduta da SETLFS e SETNAM, apre il canale di comunicazione col dispositivo precedentemente indicato nella SETLFS. È da notare che l'apertura del canale avviene senza che sia specificato se il dispositivo interessato è trasmittente o ricevente. Ad esempio se si vuole inviare al disk drive il comando di inizializzazione:

OPEN 14,8,15,"I"

la corrispondente routine in linguaggio macchina è:

```
LDA #$0E        ;OPEN 14,8,15
LDX #$08
LDY #$0F
JSR $FFBA

LDA #$00        ;nessun nome
JSR $FFBD

JSR $FFC0       ;open logical file

LDX #$0E        ;open channel for output
JSR $FFC9
```

```
LDA #$49        ;invio del carattere "I"
JSR $FFD2

JSR $FFE7       ;chiusura di tutti i file
RTS
```

Il lettore avrà notato l'uso di altre tre routine e precisamente quelle che iniziano in locazione $FFC9, $FFD2 e $FFE7 le cui funzioni sono descritte qui di seguito.

Open channel for output $FFC9

Questa routine predispone come canale di uscita quello individuato dal contenuto del registro X: nel programma precedente le istruzioni:

```
LDX #$0E
JSR $ FFC9
```

fanno sì che sia di uscita il canale associato al numero 14 ($0E), cioè quello inizialmente predisposto per il colloquio con l'unità a dischi.

Output character to channel $FFD2

Essa permette di inviare un carattere al canale predisposto come uscita: nell'esempio precedente

```
LDA #$49
JSR $FFD2
```

inviano al 1541 il carattere "I".

Close all files $FFE7

chiude tutti i canali aperti ripristinando però nelle loro funzioni quelli associati alla tastiera e al modulatore video.

Esistono inoltre due routine in qualche modo complementari alla $FFC0 e alla $FFC9:

Close logical file $FFC3

Essa chiude il canale specificato del contenuto dell'accumulatore. Ad esempio:

```
LDA #$0F
JSR $FFC3
```

chiude il canale associato al dispositivo numero 15.

Open channel for input $FFC6

Questa routine predispone in ingresso il canale individuato dal contenuto del registro X.
Associata a $FFC6 è la routine $FFCF.

Input character from channel $FFCF

Fornisce nell'accumulatore un carattere dal canale di ingresso aperto precedentemente da una chiamata alla $FFC6. Ad esempio il programma:

```
LDA #$01        ;OPEN1,1,0
LDX #$01
LDY #$00
JSR $FFBA
LDA #$00
JSR $FFBD
JSR $FFC0
LDX #$01
JSR $FFC6
JSR $FFCF       ;GET #1,A$
LDA #$01
JSR $FFC0       ;CLOSE 1
RTS
```

preleva un carattere dal primo file sequenziale trovato sul nastro.
Una routine di notevole importanza per il colloquio con le periferiche è:

Read I/O status word $FFB7

Questa fornisce nell'accumulatore il valore della variabile di stato ST; durante la lettura di un file, sia programma che sequenziale, è usata soprattutto per riconoscere se è stato letto l'ultimo carattere: in questo caso il valore presente nell'accumulatore è $40.

Questa breve carrellata sulle principali routine del KERNAL e il programma presentato dovrebbero avere messo in evidenza l'estrema utili-

tà di poter usare quanto più possibile le risorse del KERNAL; a tale scopo una descrizione abbastanza accurata di tali routine è reperibile in bibliografia.

AUTORUN

In questo paragrafo è descritto un metodo per far sì che un programma, caricato in memoria con

LOAD "nome", X, 1 (X = 1 per il nastro, X = 8 per il disco)

sia mandato in esecuzione in modo automatico cioè senza che l'operatore dia il comando RUN, o SYS nel caso il programma sia in linguaggio macchina.
Il metodo utilizzato, che come si vedrà è valido per programmi scritti sia in linguaggio macchina che in Basic, si basa su un'opportuna modifica della tabella dei vettori, quella contenuta nelle locazioni da $ 0300 a $ 030B; essa, come è noto, fornisce all'interprete Basic gli indirizzi di alcune importanti routine come quella di gestione degli errori, di tokenizzazione, di accettazione di un comando da tastiera eccetera.
Se infatti si fa in modo che un programma, caricato con indirizzo secondario pari a 1, e quindi depositato nella memoria del calcolatore a partire dalla locazione indicata nell'*header* del programma stesso, modifichi durante il suo caricamento in memoria la tabella dei vettori in modo che puntino alla locazione ove esso inizia, allora è mandato in esecuzione in modo automatico.
Questo accade perché l'interprete Basic, effettuato un LOAD, va a eseguire la routine di accettazione di comandi da tastiera, cioè quella che fa apparire sullo schermo televisivo la scritta READY e il cursore lampeggiante; l'indirizzo di tale routine è contenuto nelle locazioni $ 0302-$ 0303: in queste locazioni, durante il LOAD del programma, viene posto proprio l'indirizzo di inizio del programma così che questo viene automaticamente mandato in esecuzione.
Da quanto detto si potrebbe pensare di salvare su nastro o su disco tutta l'area di memoria RAM che inizia da $ 0300 e termina nella locazione in cui è presente l'ultimo byte del programma, avendo preventivamente e ovviamente modificato la tabella dei vettori: questo però non è possibile in quanto una semplice modifica della tabella suddetta non permette il corretto funzionamento dell'interprete Basic.
Il problema si può superare utilizzando un programma caricatore (in gergo si chiama *loader*) il quale non appena caricato va automaticamen-

te in esecuzione ed effettua l'equivalente di un LOAD + RUN del programma principale, nel caso questo sia scritto in Basic, oppure di un SYS se è in linguaggio macchina.
Nella memoria RAM del VIC-20 vi è un insieme di locazioni, da $ 02A1 a $ 02FF, che non sono utilizzate né dal KERNAL, né dall'interprete Basic; tra l'altro queste locazioni sono adiacenti a quelle in cui è contenuta la tabella dei vettori.
Il *loader* lo possiamo allora allocare in quest'area senza tema di provocare il blocco del calcolatore.
Vediamo in pratica come si può ottenere quanto ci si era proposto: cominciamo per semplicità dal problema di caricare e mandare in esecuzione in modo automatico un programma scritto in linguaggio macchina.
Per esemplificare chiamiamo il *loader* PROVA CARICATORE e PROVA MACCH. il programma principale.
Il *loader* è:

```
$02A1     LDA #$93 ; azzeramento
$02A3     JSR $FFD2; dello schermo
                   ; (non necessario)
$02A6     LDA #$02 ; OPEN 2,8,1
$02A8     LDX #$08 ; (01 per nastro)
$02AA     LDY #$01 ; indirizzo secondario
$02AC     JSR $FFBA

$02AF     LDA #$0C ; si fornisce
$02B1     LDX #$F0 ; il nome
$02B3     LDY #$02 ;
$02B5     JSR $FFBD

$02B8     LDA #$00 ; load
$02BA     STA $009D
$02BC     JSR $FFD5

$02BF     JMP $xxxx ; salta alla esecuzione
          del programma principale
```

Come si vede il programma *loader* è molto semplice; occorre però fornire, a partire dalla locazione $ 02F0 il nome del programma principale, nel nostro caso PROVA MACCH. che esso deve caricare e mandare in esecuzione.

Allora a partire dalla locazione $ 02F0 debbono essere posti i byte:

50, 52, 4F, 56, 41, 20, 4D, 41, 43, 43, 48, 2E

corrispondenti ai caratteri:

P R O V A M A C C H .

Inoltre si debbono modificare le locazioni da $ 0300 a $ 030B in modo che i vettori in esse contenuti puntino tutti a $ 02A1:

```
$0300    A1
$0301    02
$0302    A1
$0303    02
$0304    A1
$0305    02
$0306    A1
$0307    02
$0308    A1
$0309    02
$030A    A1
$030B    02
```

Come il lettore avrà certamente notato è necessario l'uso di un assemblatore col quale si dovranno scrivere le istruzioni del *loader* e modificare le locazioni $02F0... e le $0300...
Una volta fatto il necessario si salverà, sempre tramite l'assemblatore, tutta l'area di memoria compresa tra $02A1 e $030B.
Si noti che il programma principale (PROVA MACCH.) deve iniziare nella locazione $xxxx e che deve essere salvato immediatamente dopo il *loader* nel caso si usi il nastro.
Riassumiamo brevemente il funzionamento del *loader*: una volta che esso sia stato caricato nel calcolatore con

LOAD "LOADER",X,1

esso ha modificato evidentemente le locazioni da $0300 a $030B; terminato il LOAD l'interprete Basic effettua un salto alla routine che inizia in $0302 ma ora, al posto dell'indirizzo usuale $C483, esso trova $02A1 e quindi viene mandato in esecuzione il *loader* stesso il quale effettua il caricamento in memoria di PROVA MACCH. e quindi lo manda in esecuzione tramite la JMP xxxx.

Rispetto a quanto appena visto, le cose da fare nel caso il programma principale sia scritto in Basic sono un po' più complicate, infatti ora si debbono sì modificare le locazioni della tabella dei vettori per fare sì che vada in esecuzione il *loader*, ma si debbono anche ripristinare tali locazioni ai valori richiesti dall'interprete Basic affinché il programma principale possa essere eseguito.
Il *loader* in questo caso può essere:

```
$02A1   LDA #$93 ; azzeramento
$02A3   JSR $FFD2 ; dello schermo

$02A6   LDA #$02 ; OPEN 2,8,1
$02A8   LDX #$08 ; (01 per nastro)
$02AA   LDY #$01 ; indirizzo secondario
$02AC   JSR $FFBA

$02AF   LDA #$0C ; predisposizione
$02B1   LDX #$E0 ; del nome
$02B3   LDY #$02 ;
$02B5   JSR $FFBD

$02B8   LDA #$00 ; load
$02BA   STA $009D
$02BC   JSR $FFD5

$02BF   STX $2D ; aggiornamento del
$02C1   STY $2E ; puntatore END OF PROGRAM

$02C3   LDA #$0C ; ripristino della
$02C5   LDA $02EF,X ; tabella dei
$02C8   STA $02FF,X ; vettori
$02CB   DEX
$02CD   BNE $02C5

$02CF   LDA #$00 ; impone alla CHARGET
$02D1   STA $7A ; l'indirizzo 4096
$02D3   LDA #$10 ; (VIC senza espansioni
$02D5   STA $7B ; di memoria)

$02D7   JSR $C660 ; effettua un CLR
$02DA   JMP $C7ED ; salta alla routine di
                  ; esecuzione del Basic
```

Anche qui occorre scrivere a partire però dalla locazione $02E0 i byte corrispondenti al codice ASCII dei caratteri costituenti il nome del programma principale; se esso si chiamasse PROVA BASIC, tali byte sarebbero:

$02E0 : 50, 52, 4F, 56, 41, 20, 42, 41, 53, 49, 43, 2E

Le locazioni dalla $0300 alla $030B compresa vanno modificate come nel caso del *loader* per programmi in linguaggio macchina in modo che puntino a $ 02A1.
A partire dalla locazione $02F0 vanno invece depositati i valori originali della tabella dei vettori, valori che saranno letti e ripristinati dalle istruzioni LDX #$0C eccetera. I byte da scrivere sono allora:

$02F0 : 3A, C4, 83, C4, 7C, C5, 1A, C7, E4, C7, 86, CE

Anche in questo caso il *loader* deve essere salvato su nastro o su disco (con l'assemblatore) dalla locazione $ 02A1 alla $ 030B compresa.
Si noti che per le due versioni del *loader* descritte è possibile effettuare una semplice protezione anticopia: se infatti si sostituiscono le due istruzioni

```
$02A1        LDA #$93
$02A3        JSR $FFD2
```

che effettuano l'equivalente di un comando diretto SHIFT + CLR e che non sono quindi indispensabili, con:

```
$02A1        LDA #$64
$02A3        STA $0328
```

una volta che queste siano state eseguite si disabilitano le funzioni STOP + RESTORE e il comando LIST. In questo modo una volta dato il LOAD per il programma *loader* non è più molto facile indagare su come sono realizzati il *loader* stesso e il relativo programma principale.

CAPITOLO OTTO

I cunei nel sistema operativo e nell'interprete Basic

L'organizzazione del sistema operativo della Commodore è molto flessibile dato che l'indirizzo di inizio delle routine principali caratteristiche, in linguaggio macchina, è contenuto in alcune locazioni della memoria RAM che, nel loro insieme, costituiscono una tabella (tabella 8.1) cui si può fare riferimento con istruzioni di salto indiretto (JMP (...)).

Tabella 8.1 I principali vettori del sistema operativo

$0314-$0315	($EABF)	vettore di interruzione IRQ (da timer)
$0316-$0317	($FED2)	vettore di interruzione software (BRK)
$0318-$0319	($FEAD)	vettore di interruzione non mascherabile (NMI)
$031A-$031B	($F40A)	vettore per l'apertura di un canale
$031C-$031D	($F34A)	vettore per la chiusura di un canale
$031E-$031F	($F2C7)	vettore per la predisposizione di un canale per l'ingresso
$0320-$0321	($F309)	vettore per la predisposizione di un canale per l'uscita
$0322-$0323	($F3F3)	vettore per la chiusura di tutti i canali
$0324-$0325	($F20E)	vettore per l'ingresso di dati da un canale
$0328-$0329	($F3EF)	vettore di rilevazione del tasto RUN/STOP
$032A-$032B	($F1F5)	vettore per la routine di GET da tastiera
$032C-$032D	($F3EF)	vettore di chiusura di tutti i canali
$032E-$032F	($FED2)	vettore per la routine di USR
$0330-$0331	($F549)	vettore link per il LOAD
$0332-$0333	($F685)	vettore link per il SAVE

Se si modifica il contenuto di questa tabella si può quindi interagire con il sistema operativo per far eseguire al calcolatore, in risposta ad alcuni comandi o a certe situazioni, invece delle sue routine altre scritte dall'utente.
Queste modifiche, o cunei (*wedge*) possono essere inserite con facilità nel sistema operativo specialmente nella routine di gestione delle interruzioni da timer e in quella di CHARGET, che preleva i token di un'istruzione dal buffer del Basic.

CUNEI NELLA ROUTINE DI INTERRUZIONE

La routine di interruzione è mandata in esecuzione ogni sessantesimo di secondo a seguito di una richiesta generata da un timer programmato allo scopo che risiede in uno dei due VIA. A ogni interruzione il microprocessore termina l'esecuzione dell'istruzione corrente, abbandona la sequenza di istruzioni che stava eseguendo e inizia l'esecuzione di una routine allocata a partire da $FF72.
Questa routine, che tra l'altro effettua la scansione della tastiera, viene mandata in esecuzione mediante un salto (JMP indiretto) all'indirizzo contenuto nella memoria RAM nelle locazioni $0314 e $0315.
Il normale contenuto di queste locazioni è $EABF: se si varia l'indirizzo, ad esempio con POKE, è possibile fare sì che il calcolatore, ogni sessantesimo di secondo, esegua la routine o le routine che vogliamo.
È importante tenere presente che, per mantenere l'integrità del sistema operativo, bisogna, una volta eseguite le routine cuneo, far eseguire anche quelle che iniziano nella locazione $EABF.
In pratica per effettuare un cuneo nella routine di interruzione occorre:

1. modificare il contenuto delle locazioni $0314 e $0315 in modo che esso punti all'indirizzo di inizio della routine cuneo da noi inserita;
2. occorre che a partire da questo indirizzo esista la nostra routine;
3. infine al termine della routine-cuneo deve essere mandata in esecuzione la normale routine che inizia in $EABF.

Come esempio è descritta una routine cuneo la quale fa sì che nella prima riga in alto dello schermo video appaia un contatore di secondi, modulo 10.
Occorrono in pratica tre distinte routine, strettamente collegate fra loro, che debbono essere presenti contemporaneamente in memoria. La prima serve per attivare e inizializzare il cuneo stesso: non fa altro che modificare il valore delle locazioni $0314 e $0315, predisporre il

contatore a 0 e il codice di carattere che sarà visualizzato sullo schermo televisivo al valore $30, corrispondente al carattere "0".

Indirizzo	*Istruzione*	*Commento*
1DA0	SEI	;disabilitazione delle interruzioni
1DA1	LDA #$ B7	;parte meno significativa del nuovo indirizzo
1DA3	STA $0314	;della routine di interruzione in $0314
1DA6	LDA #$ 1D	;come sopra per la parte più significativa
1DA8	STA $0315	;in $0315
1DAB	LDA #$00	;inizializzazione del contatore a 0
1DAD	STA $1DF0	;il contatore è nella locazione $1DF0
1DB0	LDA #$ 30	;codice ASCII della cifra 0
1DB2	STA $1DF1	;nella locazione che contiene il carattere da inviare nello schermo
1DB5	CLI	;riabilitazione delle interruzioni
1DB6	RTS	;ritorno al Basic

Come si può notare, la routine di inizializzazione del cuneo per prima cosa disabilita le interruzioni; il motivo per cui si deve far eseguire questa disabilitazione è presto detto: si immagini che sia stata mandata in esecuzione una routine simile a quella descritta ma senza le due istruzioni SEI e CLI. Se non arriva una richiesta di interruzione da timer mentre il microprocessore sta eseguendo questa routine le cose procedono bene e il nostro cuneo è correttamente inserito.
Se invece arriva l'interruzione, ad esempio durante l'esecuzione della istruzione LDA #$ 1D, il microprocessore termina l'esecuzione della stessa, poi salta ad eseguire la routine di indirizzo $FF72: ad un certo istante questa preleverà dalle locazioni $0314 e $0315 l'indirizzo della prossima istruzione da eseguire e qui cominciano i guai! Infatti la routine di inizializzazione del cuneo prima di essere interrotta ha sì scritto in $0314 il valore $A7 ma non ha ancora modificato il contenuto di $0315 perché è stata interrotta nel mezzo della modifica.
Il risultato è che il microprocessore salterà ad eseguire la routine che parte dalla locazione $EAA7 e non certo dalla $1DA7 come si voleva: probabilmente il calcolatore si bloccherà!
Vista la necessità dell'istruzione SEI è evidente quella di CLI: con questa si ripristinano la possibilità di interruzione per permettere effettivamente le operazioni della routine di wedge vera e propria e di quelle del KERNAL.
Prima di analizzare la routine cuneo vera e propria vediamo cosa

contengono le locazioni $1DF0 e $1DF1: in $1DF0 risiede un contatore modulo 60; in ogni istante essa contiene un valore pari al numero di interruzioni avvenute fino a quel momento; il conteggio è modulo 60, quindi il contenuto di tale locazione si dovrà aggiornare secondo la sequenza 0, 1, 2, 3, ..., 58, 59, 0, 1, eccetera.
In $1FD1 è invece contenuto il codice ASCII che corrisponde al carattere da visualizzare: quindi $30 se questo è lo zero, $31 se è l'uno e così via.
Per quanto concerne la routine cuneo vera e propria essa inizia per nostra scelta in $1DB7 e infatti le precedenti istruzioni:

```
LDA #$B7
STA $0314
LDA #$1D
STA $0315
```

hanno modificato il vettore presente in $0314 e in $0315 dal valore originario $EABF proprio al valore $1DB7.
La routine inizia in $1DB7, ogni sessantesimo di secondo, che è il periodo con cui è mandata in esecuzione, incrementa il contatore presente nella locazione $1DF0 e verifica se è arrivato al valore 60 (= $3C).
Se ciò è vero viene incrementato il valore presente nella locazione $1DF1, altrimenti si salta alla normale routine di interruzione.
Se è stato incrementato il valore presente in $1DF1 si verifica se esso ha superato il valore $39 che corrisponde, sul video, al carattere "9". Se ciò è accaduto in $1DF1 è immesso un'altra volta il valore iniziale $30.
Qualsiasi siano state le condizioni presentatesi in $1DF0 e in $1DF1, la routine trasferisce nella locazione $1E10, di schermo, il valore contenuto in $1DF1, e nella locazione della memoria di colore $9610 il valore $00 che corrisponde al colore nero (si è nell'ipotesi che il calcolatore non abbia collegati moduli di RAM aggiuntivi).
Infine l'ultima istruzione del cuneo è quella di salto alla locazione $EABF, cioè alla routine originaria del sistema operativo.

Indirizzo	*Istruzione*	*Commento*
1DB7	INC $1DF0	;incremento contatore
1DBA	LDA $1DF0	;il contatore è arrivato
1DBD	CMP #$3C	;al valore 60?
1DBF	BNE $1DC6	;no: salta ad aggiornare il carattere
1DC1	LDA #$00	;sì: azzera il contatore
1DC3	STA $1DF0	
1DC6	INC $1DF1	;aggiorna la locazione
1DC9	LDA $1DF1	;che contiene il codice ASCII

```
1DCC   CMP #$3A    ;ha superato il carattere "9"
1DCE   BNE $1DD5   ;no: prosegui
1DD0   LDA #$30    ;sì: scrivi il codice del carattere "0"
1DD2   STA $1DF1
1DD5   STA $1E10   ;scrivi il codice nella RAM video
1DD8   LDA #$00    ;scrivi in colore nero
1DDA   STA $9610
1DDD   JMP $EABF   ;salta alla normale routine di interruzione
```

Le due routine appena descritte permettono di inserire il cuneo con l'istruzione Basic **SYS 7584**: non appena questa è eseguita, o da programma o in modo diretto, appare il contasecondi nella undicesima colonna dello schermo televisivo e vi permane fino a che non si attivano contemporaneamente i tasti **RUNSTOP** e **RESTORE**.
Se si vuole invece ripristinare la situazione normale senza il **RESTORE** e quindi far sparire il contasecondi, occorre disattivare il cuneo mandando in esecuzione la routine seguente che pone in **$0314** e in **$0315** i valori originari; tale routine è mandata in esecuzione da **SYS7664** che può essere fatta eseguire sia da programma che in modo diretto.

```
1DE0     SEI
1DE1     LDA #$BF
1DE3     STA $0314
1DE6     LDA #$EA
1DE8     STA $0315
1DEB     CLI
1DEC     RTS
```

CUNEI NELLA ROUTINE "CHARGET"

La routine di CHARGET risiede nella memoria RAM nelle locazioni da **$0073** a **$008A** compresa; essa costituisce il "trait d'union" tra un programma Basic, o un comando diretto da tastiera, e l'interprete Basic stesso.
Come abbiamo visto un comando diretto da tastiera è depositato, dopo essere stato trasformato in token, in un buffer di ingresso per il Basic che è allocato nelle locazioni da **$0200** a **$0257**.
La routine di CHARGET ha un modo di operare molto semplice: essa scandisce il contenuto del buffer di ingresso per il Basic, ignorando i codici corrispondenti al carattere "spazio" (=**$20**), e deposita nell'accu-

mulatore del microprocessore i byte che via via incontra, settando nel contempo il flag di carry se il byte trovato non corrisponde al codice ASCII di un carattere numerico, resettandolo in caso contrario.
A questo punto, cioè dopo ogni deposito di un byte nell'accumulatore, entra in funzione l'interprete Basic che agisce in conseguenza.
La routine di CHARGET è listata di seguito:

Indirizzo	*Istruzione*
0073	INC $7A
0075	BNE $0079
0077	INC $7B
0079	LDA $xxxx
007C	CMP #$3A
007E	BCS $008A
0080	CMP #$20
0082	BEQ $0073
0084	SEC
0085	SBC #$30
0087	SEC
0088	SBC #$D0
008A	RTS

Il valore xxxx è quello contenuto nelle locazioni $007A e $007B che sono continuamente aggiornate dalla routine stessa per effettuare la scansione del buffer di ingresso o del programma per l'interprete Basic.
Per aggiungere un nuovo comando al Basic mediante un cuneo nella routine CHARGET occorre modificarla in modo che:

a. riesca a rivelare la presenza del token corrispondente al nuovo comando;
b. se esso è presente ne mandi in esecuzione la routine corrispondente, altrimenti esegua i suoi normali compiti.

In pratica conviene associare il o i nuovi comandi a un simbolo della tastiera, preferibilmente uno di quelli poco usati, ad esempio il "&", il cui codice ASCII è $26, e fare sì che la "nuova" CHARGET, se lo rileva, mandi in esecuzione la routine corrispondente, la quale dovrà poi a sua volta risaltare alla prima istruzione della CHARGET stessa. Quest'ultima azione fa sì che l'interprete Basic non si accorga minimamente della presenza del carattere "speciale".
Come esempio si dà una semplice routine, associata al carattere &, la quale fa sì che venga emesso dal televisore un "beep" ogni volta che

viene premuto il tasto relativo oppure ogni volta che appare nel programma il carattere &.

Inizializzazione del cuneo

La routine di CHARGET è modificata nelle locazioni $0084 e $0086 in modo che diventi:

```
....
....
0084     JMP $xxxx
....
....
```

dove $xxxx è l'indirizzo del nostro cuneo.
Occorre allora costruire una routine che effettui questa modifica, e la allochiamo a partire dalla locazione $1D00:

```
1D00  SEI
1D01  LDA #$4C   ; (codice di JMP)
1D03  STA $0084
1D06  LDA #$10
1D08  STA $0085
1D0B  LDA #$1D
1D0D  STA $0086
1D10  CLI
1D11  RTS
```

Essa non fa altro che scrivere nella locazione $0084 l'istruzione JMP $1D10 che effettua il salto alla prima istruzione del programma che rivela la presenza del simbolo "&".
A partire da $1D10 è memorizzata la routine cuneo vera e propria:

Indirizzo	*Istruzione*	*Commento*
1D10	CMP #$26	;il token è "&"?
1D12	BNE $1D2F	;se no salta alla normale routine CHARGET
1D14	LDA #$0F	;volume al massimo
1D16	STA $900F	
1D19	LDA #$F0	;240 al generatore di
1D1B	STA $900D	;toni alti

```
1D1E    LDX #$FF     ;loop di attesa
1D20    LDY #$50
1D22    DEY
1D23    BNE $1D22
1D25    DEX          ;fine del loop
1D26    BNE $1D20
1D28    LDA #$00     ;volume a zero
1D2A    STA $900F
1D2D    JMP $0073    ;salto all'inizio di CHARGET
1D2F    SEC          ;ripristino delle due istruzioni
1D30    SBC #$30     ;sostituite nella CHARGET dal cuneo
1D32    JMP $0087    ;salto al proseguimento della CHARGET
```

CUNEI NELL'INTERPRETE BASIC

Finora si sono visti due modi per interagire col sistema operativo e con l'interprete Basic che in pratica hanno realizzato un cuneo nel sistema operativo (il contasecondi) e un nuovo comando Basic: l'"&", che genera un beep.
Sorge spontaneo porsi la domanda se sia possibile fare di più; la risposta è affermativa nel senso che si possono introdurre nuovi comandi Basic molto più potenti inserendo cunei proprio nell'interprete Basic.
La procedura per fare ciò è un po' complicata dato che occorre modificare radicalmente la routine che effettua la tokenizzazione, quella di LIST, o di detokenizzazione, quella di rilevazione di errori di sintassi, quella di individuazione di un token e quella di valutazione di una espressione numerica.
Ma come è possibile modificare tali routine dato che esse risiedono, come si è visto, nella ROM? La risposta è che esse sì risiedono in ROM ma la loro chiamata avviene tramite una tabella di indirizzi che risiede in RAM (tabella 8.2), come si è già visto nell'esempio del contasecondi.
In questo caso basterà modificare i contenuti di un certo numero di locazioni in tale tabella di modo che il sistema operativo o l'interprete Basic vadano prima a eseguire le nostre routine cuneo e poi quelle normali.
Ad esempio si vuole creare il nuovo comando Basic:

SOUND (X, Y, Z, W, V)

il quale immette nei generatori di suono basso, medio, alto e di rumore rispettivamente i valori X, Y, Z, W (X, Y, Z, W maggiori o uguali a 128 e

Tabella 8.2 Le principali routine dell'interprete Basic

$0300-$0301	($C43A)	link alla routine dei messaggi di errore
$0302-$0303	($C483)	link alla routine principale dell'interprete Basic
$0304-$0305	($C57C)	link alla routine di tokenizzazione (ingresso da tastiera)
$0306-$0307	($C71A)	link alla routine di detokenizzazione (per il LIST)
$0308-$0309	($C7E7)	link alla routine di individuazione del token e di lancio del relativo programma
$030A-$030B	($CE86)	link alla routine di calcolo di un'espressione numerica

minori di 256), e predispone il volume al valore V (maggiore o uguale a 0 e minore di 16).
Per fare sì che l'interprete Basic riconosca il nuovo comando occorre:

1. Scrivere una routine, mandata in esecuzione dal comando SOUND (....), che si aspetti in successione cinque valori decimali separati da una virgola e seguiti dal simbolo ")", e predisponga i generatori di suono e il volume secondo i cinque parametri. La routine deve inoltre verificare la correttezza del nuovo comando Basic e nel caso l'operatore abbia commesso un errore di sintassi, dare il messaggio SYNTAX ERROR.
2. Modificare la routine di tokenizzazione dei comandi Basic in modo che riesca a tokenizzare la stringa SOUND(.
3. Modificare la routine di listing in modo che durante il listato del programma, a ogni occorrenza dei token corrispondente al nuovo comando, generi la stringa di caratteri SOUND(.

Il seguente programma in linguaggio macchina ci permette di realizzare questo cuneo:

```
                   1               ORG    $1C00
                   2
1C00: EA           3               NOP
1C01: EA           4               NOP
1C02: EA           5               NOP
1C03: EA           6               NOP
1C04: EA           7               NOP
1C05: EA           8               NOP
1C06: EA           9               NOP
1C07: EA           10              NOP
1C08: EA           11              NOP
                   12
1C09: 20 3F 1C     13    START     JSR    H1C3F      ; RESETTA IL CALCOLATORE
1C0C: 20 13 1C     14              JSR    VETTORI    ; INIZIALIZZA I VETTORI
1C0F: 58           15              CLI
1C10: 4C 7B E3     16              JMP    $E37B      ;INIZIALIZZA IL RESTO
                   17
```

```
1C13: A2 ØB       18   VETTORI  LDX   #$ØB
1C15: BD 1F 1C    19   VETLOOP  LDA   TABVETT,X
1C18: 9D ØØ Ø3    2Ø            STA   IERROR,X
1C1B: CA          21            DEX
1C1C: 1Ø F7       22            BPL   VETLOOP
1C1E: 6Ø          23            RTS
                  24
1C1F: 3A C4       25   TABVETT  DA    $C43A      ;MESSAGGI DI ERRORE
                  26
1C21: 83 C4       27            DA    $C483      ;INTERPRETA ISTRUZIONI
                  28
1C23: 81 1C       29            DA    $1C81      ;TRADUCE KEYWORDS IN TOKENS
                  3Ø
1C25: E3 1C       31            DA    $1CE3      ;TRADUCE TOKENS IN KEYWORDS
                  32
1C27: 16 1D       33            DA    $1D16      ;INTERPRETA I NUOVI COMANDI
                  34
1C29: 3C 1D       35            DA    $1D3C      ;VALUTA UNA  ESPRESSIONE
                  36
                  37
1C2B: 2C 11 91    38            BIT   $9111      ;GESTISCE RUN/STOP RESTORE
1C2E: 2Ø 34 F7    39            JSR   $F734
1C31: 2Ø E1 FF    4Ø            JSR   STOP
1C34: DØ Ø6       41            BNE   FINE
1C36: 2Ø 42 1C    42            JSR   INVETT
1C39: 6C Ø2 CØ    43            JMP   ($CØØ2)    ;RITORNA AL BASIC
                  44
1C3C: 4C 56 FF    45   FINE     JMP   $FF56      ;RITORNA AL S.O.
1C3F: 2Ø 8D FD    46   H1C3F    JSR   $FD8D      ;TESTA LA RAM
                  47
                  48
1C42: 2Ø 8A FF    49   INVETT   JSR   RESTOR     ;INIZIALIZZA I VETTORI
                  5Ø                             ;DEL S.O.
1C45: 2Ø F9 FD    51            JSR   $FDF9      ;INIZIALIZZA L' I/O
1C48: 2Ø 18 E5    52            JSR   $E518      ;INIZIALIZZA LO SCHERMO
1C4B: A9 B9       53            LDA   #$B9
1C4D: 8D 16 Ø3    54            STA   CBINV
1C5Ø: A9 A2       55            LDA   #$A2
1C52: 8D 17 Ø3    56            STA   $Ø317
1C55: 6Ø          57            RTS
1C56: 64 1D       58   WEDGE    DA    SOUND-1
1C58: EA EA EA    59            HEX   EAEAEAEAEAEAEAEAEAEA
1C5B: EA EA EA EA EA EA EA
1C62: 53 4F 55    6Ø   KEYSOUND DCI   'sound('
1C65: 4E 44 A8
1C68: ØØ          61            HEX   ØØ
                  62
                  63
                  64            ORG   $1C81
                  65
1C81: 2Ø 7C C5    66            JSR   $C57C      ;NUOVA ROUTINE DI TOKENIZZAZIONE
1C84: AØ Ø5       67            LDY   #$Ø5
1C86: B9 FB Ø1    68   H1C86    LDA   BUFF,Y
1C89: FØ 57       69            BEQ   H1CE2
1C8B: C9 22       7Ø            CMP   #$22
1C8D: FØ 47       71            BEQ   H1CD6
1C8F: C9 41       72            CMP   #$41
1C91: 9Ø 4Ø       73            BCC   H1CD3
1C93: C9 5B       74            CMP   #$5B
1C95: BØ 3C       75            BCS   H1CD3
1C97: 84 B1       76            STY   $B1
1C99: A2 ØØ       77            LDX   #$ØØ
1C9B: 86 ØB       78            STX   $ØB
1C9D: 38          79   H1C9D    SEC
1C9E: FD 62 1C    8Ø            SBC   KEYSOUND,X
```

```
1CA1: F0 13      81            BEQ   H1CB6
1CA3: C9 80      82            CMP   #$80
1CA5: F0 16      83            BEQ   H1CBD
1CA7: BD 62 1C   84   H1CA7    LDA   KEYSOUND,X
1CAA: F0 27      85            BEQ   H1CD3
1CAC: 30 03      86            BMI   H1CB1
1CAE: E8         87            INX
1CAF: D0 F6      88            BNE   H1CA7
1CB1: E6 0B      89   H1CB1    INC   $0B
1CB3: A4 B1      90            LDY   $B1
1CB5: A9 C8      91            LDA   #$C8
1CB7: B9 FB 01   92            LDA   BUFF,Y
1CBA: E8         93            INX
1CBB: D0 E0      94            BNE   H1C9D
1CBD: A6 B1      95   H1CBD    LDX   $B1
1CBF: A5 0B      96            LDA   $0B
1CC1: 18         97            CLC
1CC2: 69 CC      98            ADC   #$CC
1CC4: 9D FB 01   99            STA   BUFF,X
1CC7: C8         100  H1CC7    INY
1CC8: E8         101           INX
1CC9: B9 FB 01   102           LDA   BUFF,Y
1CCC: 9D FB 01   103           STA   BUFF,X
1CCF: D0 F6      104           BNE   H1CC7
1CD1: A4 B1      105           LDY   $B1
1CD3: C8         106  H1CD3    INY
1CD4: D0 B0      107           BNE   H1C86
1CD6: C8         108  H1CD6    INY
1CD7: B9 FB 01   109           LDA   BUFF,Y
1CDA: F0 06      110           BEQ   H1CE2
1CDC: C9 22      111           CMP   #$22
1CDE: D0 F6      112           BNE   H1CD6
1CE0: F0 F1      113           BEQ   H1CD3
1CE2: 60         114  H1CE2    RTS
1CE3: 08         115           PHP
                 116
                 117
1CE4: C9 FF      118           CMP   #$FF          :NUOVA ROUTINE DI LIST
1CE6: F0 2A      119           BEQ   H1D12
1CE8: 24 0F      120           BIT   $0F
1CEA: 30 26      121           BMI   H1D12
1CEC: C9 CC      122           CMP   #$CC
1CEE: 90 22      123           BCC   H1D12
1CF0: 28         124           PLP
1CF1: 38         125           SEC
1CF2: E9 CB      126           SBC   #$CB
1CF4: AA         127           TAX
1CF5: 84 49      128           STY   $49
1CF7: A0 FF      129           LDY   #$FF
1CF9: CA         130  H1CF9    DEX
1CFA: F0 08      131           BEQ   H1D04
1CFC: C8         132  H1CFC    INY
1CFD: B9 62 1C   133           LDA   KEYSOUND,Y
1D00: 10 FA      134           BPL   H1CFC
1D02: 30 F5      135           BMI   H1CF9
1D04: C8         136  H1D04    INY
1D05: B9 62 1C   137           LDA   KEYSOUND,Y
1D08: 30 05      138           BMI   H1D0F
1D0A: 20 D2 FF   139           JSR   CHROUT
1D0D: D0 F5      140           BNE   H1D04
1D0F: 4C EF C6   141  H1D0F    JMP   $C6EF
1D12: 28         142  H1D12    PLP
1D13: 4C 1A C7   143           JMP   $C71A
                 144
                 145
```

```
1D16: 20 73 00  146           JSR   $0073      ;NUOVA ROUTINE DI INTERPRETAZIONE
 DEI TOKENS
1D19: C9 CC     147           CMP   #$CC
1D1B: 90 19     148           BCC   H1D36
1D1D: C9 D0     149           CMP   #$D0
1D1F: B0 15     150           BCS   H1D36
1D21: 20 27 1D  151           JSR   H1D27
1D24: 4C AE C7  152           JMP   $C7AE
1D27: E9 CB     153  H1D27    SBC   #$CB
1D29: 0A        154           ASL
1D2A: A8        155           TAY
1D2B: B9 57 1C  156           LDA   WEDGE+1,Y
1D2E: 48        157           PHA
1D2F: B9 56 1C  158           LDA   WEDGE,Y
1D32: 48        159           PHA
1D33: 4C 73 00  160           JMP   $0073
                161
1D36: 20 79 00  162  H1D36    JSR   $0079      ;RIPRENDE IL CARATTERE E
1D39: 4C E7 C7  163           JMP   $C7E7      ;E TORNA ALLA INTERPRETAZIONE
                164                            ;NORMALE
                165
                166
1D3C: A9 00     167           LDA   #$00       ;VALUTAZIONE DI ESPRESSIONE
1D3E: 85 08     168           STA   $08
1D40: 20 73 00  169           JSR   $0073
1D43: C9 D0     170           CMP   #$D0
1D45: 90 13     171           BCC   H1D5A
1D47: C9 D2     172           CMP   #$D2
1D49: B0 0F     173           BCS   H1D5A
1D4B: E9 CB     174           SBC   #$CB
1D4D: 0A        175           ASL
1D4E: A8        176           TAY
1D4F: B9 66 1D  177           LDA   SOUND+1,Y
1D52: 48        178           PHA
1D53: B9 56 1C  179           LDA   WEDGE,Y
1D56: 48        180           PHA
1D57: 4C 73 00  181           JMP   $0073
1D5A: A5 7A     182  H1D5A    LDA   $7A
1D5C: D0 02     183           BNE   H1D60
1D5E: C6 7B     184           DEC   $7B
1D60: C6 7A     185  H1D60    DEC   $7A
1D62: 4C 86 CE  186           JMP   $CE86
                187
                188
1D65: A6 00     189  SOUND    LDX   $00        ;SOUND(X,Y,Z,W,U)
1D67: 8E 00 01  190  SLOOP    STX   STACK
1D6A: 20 9E D7  191           JSR   $D79E
1D6D: A8        192           TAY
1D6E: 8A        193           TXA
1D6F: AE 00 01  194           LDX   STACK
1D72: 9D 01 01  195           STA   STACK+1,X
1D75: E8        196           INX
1D76: E0 06     197           CPX   #$06
1D78: B0 0B     198           BCS   SYNTER
1D7A: 20 73 00  199           JSR   $0073
1D7D: C0 29     200           CPY   #$29
1D7F: F0 07     201           BEQ   VCRGMOV
1D81: C0 2C     202           CPY   #$2C
1D83: F0 E2     203           BEQ   SLOOP
1D85: 4C 08 CF  204  SYNTER   JMP   $CF08
1D88: CA        205  VCRGMOV  DEX
1D89: BD 01 01  206  MOVLOOP  LDA   STACK+1,X
1D8C: 9D 0A 90  207           STA   $900A,X
1D8F: CA        208           DEX
1D90: 10 F7     209           BPL   MOVLOOP
1D92: 60        210           RTS
```

```
211
212  STACK    =    $0100
213  BUFF     =    $01FB
214  IERROR   =    $0300
215  CBINV    =    $0316
216  H1CB6    =    $1CB6
217  H1DC9    =    $1DC9
218  RESTOR   =    $FF8A
219  CHROUT   =    $FFD2
220  STOP     =    $FFE1

--End assembly, 379 bytes, Errors: 0

Symbol table - alphabetical order:

   BUFF    =$01FB      CBINV   =$0316      CHROUT  =$FFD2      FINE    =$1C3C
   H1C3F   =$1C3F      H1C86   =$1C86      H1C9D   =$1C9D      H1CA7   =$1CA7
   H1CB1   =$1CB1      H1CB6   =$1CB6      H1CBD   =$1CBD      H1CC7   =$1CC7
   H1CD3   =$1CD3      H1CD6   =$1CD6      H1CE2   =$1CE2      H1CF9   =$1CF9
   H1CFC   =$1CFC      H1D04   =$1D04      H1D0F   =$1D0F      H1D12   =$1D12
   H1D27   =$1D27      H1D36   =$1D36      H1D5A   =$1D5A      H1D60   =$1D60
?  H1DC9   =$1DC9      IERROR  =$0300      INVETT  =$1C42      KEYSOUND=$1C62
   MOVLOOP =$1D89      RESTOR  =$FF8A      SLOOP   =$1D67      SOUND   =$1D65
   STACK   =$0100   ?  START   =$1C09      STOP    =$FFE1      SYNTER  =$1D85
   TABVETT =$1C1F      VCRGMOV =$1D88      VETLOOP =$1C15      VETTORI =$1C13
   WEDGE   =$1C56

Symbol table - numerical order:

   STACK   =$0100      BUFF    =$01FB      IERROR  =$0300      CBINV   =$0316
?  START   =$1C09      VETTORI =$1C13      VETLOOP =$1C15      TABVETT =$1C1F
   FINE    =$1C3C      H1C3F   =$1C3F      INVETT  =$1C42      WEDGE   =$1C56
   KEYSOUND=$1C62      H1C86   =$1C86      H1C9D   =$1C9D      H1CA7   =$1CA7
   H1CB1   =$1CB1      H1CB6   =$1CB6      H1CBD   =$1CBD      H1CC7   =$1CC7
   H1CD3   =$1CD3      H1CD6   =$1CD6      H1CE2   =$1CE2      H1CF9   =$1CF9
   H1CFC   =$1CFC      H1D04   =$1D04      H1D0F   =$1D0F      H1D12   =$1D12
   H1D27   =$1D27      H1D36   =$1D36      H1D5A   =$1D5A      H1D60   =$1D60
   SOUND   =$1D65      SLOOP   =$1D67      SYNTER  =$1D85      VCRGMOV =$1D88
   MOVLOOP =$1D89   ?  H1DC9   =$1DC9      RESTOR  =$FF8A      CHROUT  =$FFD2
   STOP    =$FFE1
```

```
10 REM*****************
11 REM*  INSERZIONE   *
12 REM*SOUND-WEDGE    *
13 REM*****************
14 :
20 POKE56,27:POKE52,27:POKE51,255:POKE55,255:CLR
30 FORI=7168TO7570
40 READA%:POKEI,A%
50 NEXT
60 SYS28*256:NEW
60000 DATA 234,234,234,234,234,234,234,234
60010 DATA 234,032,063,028,032,019,028,088
60020 DATA 076,123,227,162,011,189,031,028
60030 DATA 157,000,003,202,016,247,096,058
60040 DATA 196,131,196,129,028,227,028,022
60050 DATA 029,060,029,044,017,145,032,052
60060 DATA 247,032,225,255,208,006,032,066
60070 DATA 028,108,002,192,076,086,255,234
60080 DATA 234,234,032,138,255,032,249,253
60090 DATA 032,024,229,169,185,141,022,003
```

```
60100 DATA 169,162,141,023,003,096,100,029
60110 DATA 234,234,234,234,234,234,234,234
60120 DATA 234,234,083,079,085,078,068,168
60130 DATA 000,245,245,245,245,245,245,245
60140 DATA 245,245,245,245,245,245,245,245
60150 DATA 245,245,245,245,245,245,245,245
60160 DATA 245,032,124,197,160,005,185,251
60170 DATA 001,240,087,201,034,240,071,201
60180 DATA 065,144,064,201,091,176,060,132
60190 DATA 177,162,000,134,011,056,253,098
60200 DATA 028,240,019,201,128,240,022,189
60210 DATA 098,028,240,039,048,003,232,208
60220 DATA 246,230,011,164,177,169,200,185
60230 DATA 251,001,232,208,224,166,177,165
60240 DATA 011,024,105,204,157,251,001,200
60250 DATA 232,185,251,001,157,251,001,208
60260 DATA 246,164,177,200,208,176,200,185
60270 DATA 251,001,240,006,201,034,208,246
60280 DATA 240,241,096,008,201,255,240,042
60290 DATA 036,015,048,038,201,204,144,034
60300 DATA 040,056,233,203,170,132,073,160
60310 DATA 255,202,240,008,200,185,098,028
60320 DATA 016,250,048,245,200,185,098,028
60330 DATA 048,005,032,210,255,208,245,076
60340 DATA 239,198,040,076,026,199,032,115
60350 DATA 000,201,204,144,025,201,208,176
60360 DATA 021,032,039,029,076,174,199,233
60370 DATA 203,010,168,185,087,028,072,185
60380 DATA 086,028,072,076,115,000,032,121
60390 DATA 000,076,231,199,169,000,133,008
60400 DATA 032,115,000,201,208,144,019,201
60410 DATA 210,176,015,233,203,010,168,185
60420 DATA 087,028,072,185,086,028,072,076
60430 DATA 115,000,165,122,208,002,198,123
60440 DATA 198,122,076,134,206,162,000,142
60450 DATA 000,001,032,158,215,168,138,174
60460 DATA 000,001,157,001,001,232,224,006
60470 DATA 176,011,032,115,000,192,041,240
60480 DATA 007,192,044,240,226,076,008,207
60490 DATA 202,189,001,001,157,010,144,202
60500 DATA 016,247,096,247,008,255,008,251

READY.
```

CAPITOLO NOVE

Un convertitore analogico/digitale

PREMESSA

Prima di addentrarci nella descrizione del convertitore conviene premettere alcune nozioni fondamentali su come è possibile far colloquiare un calcolatore col mondo esterno.
Come già accennato nel capitolo 1 tutte le risorse interne al calcolatore sono gestite dal microprocessore tramite opportuni segnali che si propagano su tre bus: di indirizzi, di dati e di controllo. Dato che il solo dispositivo con il completo controllo del sistema è il microprocessore, quando esso ha bisogno di scambiare informazioni, ad esempio leggere il contenuto di una locazione di ROM o RAM, deve selezionare quella particolare locazione, informare il dispositivo che la contiene su quale è l'operazione, di lettura o di scrittura, che verrà eseguita, e infine attuare l'operazione stessa.
La selezione di un particolare dispositivo, rispetto agli altri presenti nel calcolatore, avviene tramite l'invio da parte del microprocessore di una configurazione di bit nelle linee che costituiscono il bus degli indirizzi.
La configurazione presente nel bus degli indirizzi individua in modo univoco ogni locazione di memoria possibile: dato che il 6502 può gestire al massimo 16 linee di indirizzo, esso può scegliere $2^{16} = 65536$ locazioni distinte in memoria.
Una volta presente in modo stabile un particolare indirizzo è compito

del dispositivo interessato allo scambio di informazioni col microprocessore mettersi per così dire in stato di allerta, pronto cioè o a inviare informazioni al 6502 o a riceverne. È a questo punto che il microprocessore attiva una opportuna linea del bus di controllo per effettuare l'operazione di lettura o di scrittura.
Qualsiasi sia il tipo di operazione, di lettura o di scrittura, un byte viene trasmesso al destinatario attraverso il bus dei dati.
Nello scambio di informazioni con locazioni di memoria, dato che queste sono presenti in numero notevole su di uno stesso circuito integrato, è necessario decodificare il contenuto del bus degli indirizzi per individuare quel particolare circuito integrato di memoria rispetto ad altri presenti nel calcolatore. Esistono perciò in ogni sistema a microprocessore dei dispositivi, detti decodificatori, i quali generano, in base all'indirizzo, dei segnali di selezione per i vari circuiti di memoria.

IL CONVERTITORE ANALOGICO/DIGITALE

Il convertitore qui descritto è a 8 canali: esso permetterà al VIC-20 di acquisire dati esterni di tipo analogico, come ad esempio valori di temperatura, di tensione, di intensità luminosa ecc.
Il convertitore è collegato al VIC-20 tramite il bus di espansione: ciò permette un eventuale uso della porta di utente per altri scopi e inoltre non limita in alcun modo la possibilità di aggiungere eventuali espansioni di memoria RAM o ROM tramite un connettore aggiuntivo che può essere previsto nella scheda del convertitore.

SCHEMA ELETTRICO

Lo schema del circuito è illustrato in figura 9.1; in essa si può notare un circuito di interfaccia tra il convertitore A/D e il bus di espansione.
Questo circuito ha il compito di rendere compatibili i segnali di controllo del microprocessore 6502 con quelli necessari al convertitore ADC0809.

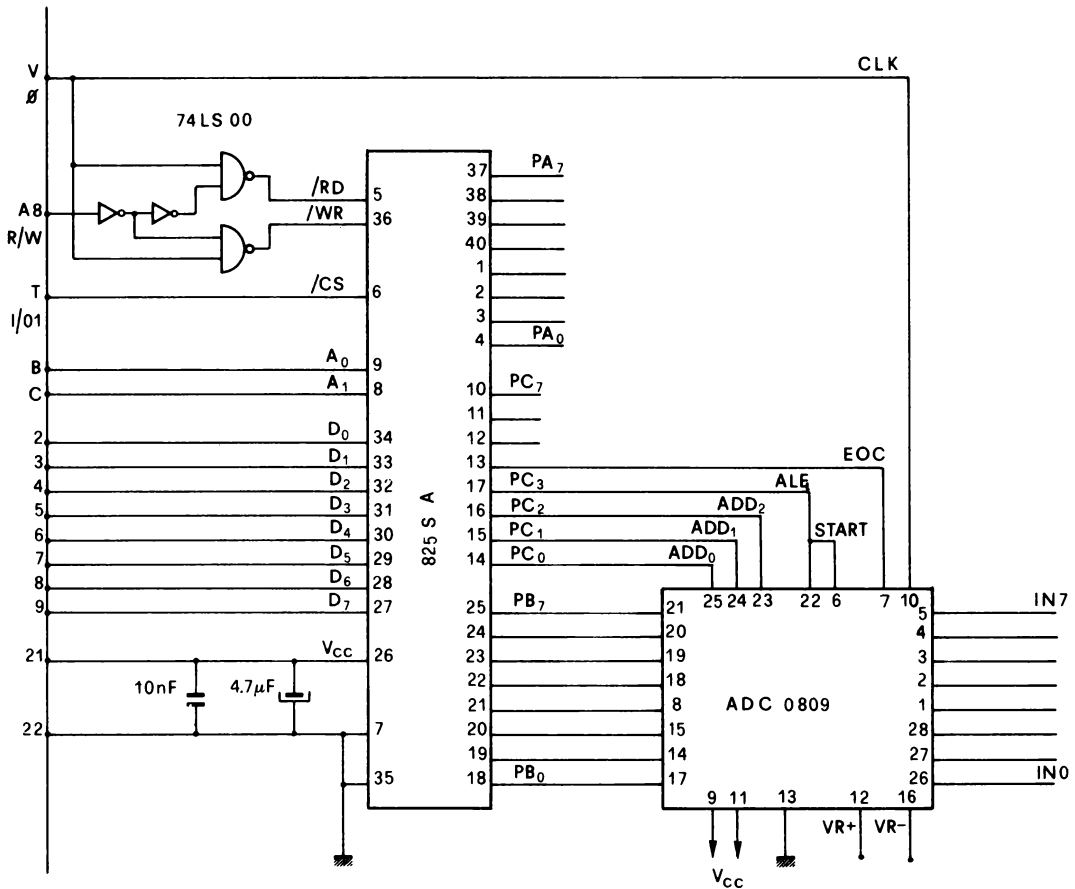



Fig. 9.1 Schema del circuito convertitore analogico/digitale.

CIRCUITO DI INTERFACCIA

L'interfaccia è costituita dal circuito TTL 74LS00 e dalla porta programmabile 8255A.
Il circuito integrato 74LS00 è utilizzato per permettere una corretta gestione della 8255A da parte del microprocessore 6502: quest'ultimo genera infatti dei segnali di controllo che non si possono applicare direttamente alla porta programmabile.
Alla 8255A sono necessari rispettivamente per la lettura e per la scrittura due segnali di controllo distinti, /RD e /WR (il segno / significa che essi sono attivi a livello logico basso, cioè con tensione $\leq$ 0,4 volt).
Il microprocessore 6502 genera invece, per i medesimi scopi, due segnali di controllo, detti R/W e φ (phi); per essere più precisi durante una

Tabella 9.1

φ	R/W	/RD	/WR
0	0	1	1
0	1	1	1
1	0	1	0
1	1	0	1

lettura i segnali R/W e φ sono ambedue a livello logico alto; durante una scrittura il primo è basso, il secondo alto.
La tabella 9.1 indica qual è il valore dei segnali /RD e /WR, generati dal 74LS00, in funzione dei segnali R/W e φ (phi) forniti dal microprocessore 6502.
Per segnalare qual è il dispositivo interessato a una operazione di lettura o di scrittura il microprocessore 6502 pone nel bus degli indirizzi quello associato dall'hardware a quel dispositivo.
Nell'interno del VIC-20 è presente un circuito decodificatore che genera un segnale al valore logico "0" non appena l'indirizzo della locazione di memoria interessata allo scambio di informazioni è compreso tra i valori 38912 e 39935 inclusi e quindi un totale di 1024 indirizzi diversi.
Tale segnale, detto /I/O BLOCK 2, è presente nel piedino T del connettore di espansione ed è collegato direttamente, nello schema, all'ingresso di selezione, o di abilitazione, /CS, dell'8255A per "avvisarlo" che l'operazione di trasferimento di dati in corso lo interessa direttamente.
Gli altri segnali necessari al funzionamento dell'8255A sono quelli del bus dei dati e del bus degli indirizzi.
I primi, indicati in figura 9.1 con D0-D7, costituiscono il bus attraverso il quale passano e le informazioni necessarie alla programmazione dell'8255A e i risultati delle conversioni analogico-digitali effettuate dall'ADC0809, oltre alle informazioni da inviare alle porte di uscita.
I segnali A0 e A1, che costituiscono le due linee meno significative del bus degli indirizzi, sono inviati direttamente agli ingressi omonimi dell'8255A: essi sono necessari in quanto tale dispositivo ha nel suo interno ben quattro unità funzionali distinte, descritte in seguito, che debbono essere individuate in modo univoco dal microprocessore stesso.
In pratica, mentre per i segnali di controllo dell'8255A è necessaria la rete realizzata dal 74LS00, i segnali di indirizzo, di dati e di abilitazione (/CS) sono compatibili direttamente a quelli generati dal microprocessore 6502.

L'8255A

Questo dispositivo, programmabile, mette a disposizione del sistema a microprocessore cui è connesso, tre porte di ingresso-uscita per il collegamento col mondo esterno. Si è detto di "ingresso-uscita" in quanto, inviando opportuni comandi di "programmazione", è possibile informare l'8255A su quale deve essere il tipo di porta (di ingresso o di uscita) che esso deve realizzare e gestire.
Le tre porte dell'8255A sono contraddistinte con i nomi porta A, porta B, porta C: si può imporre, via software, che una porta sia di ingresso, un'altra di uscita ecc.
La programmazione dell'8255A si effettua inviando byte di valore opportuno a una speciale unità funzionale, detta porta di controllo (CNTL), la quale interpreta i byte inviatile dal microprocessore come comandi di programmazione.
Senza entrare nel merito di questo argomento, per il quale si rimanda all'appendice, per il nostro scopo ci basta sapere che le tre porte: A, B, C e quella di controllo sono individuate dai seguenti indirizzi:

Indirizzo	*Porta*
38912	A
38913	B
38914	C
38915	CNTL

Per quanto riguarda la programmazione da adottare per il nostro schema essa deve essere:

Porta	*Direzione*
A	uscita
B	ingresso
C(0-3)	ingresso
C(4-7)	uscita

(la porta A non è necessaria al convertitore A/D e può essere adoperata liberamente come porta di uscita o di ingresso secondo le necessità dell'utilizzatore, cambiando evidentemente la programmazione dell'8255A).
Vediamo ora in dettaglio quali sono i collegamenti tra il convertitore A/D e l'8255A.

La porta B di quest'ultimo è dedicata al trasferimento del byte ottenuto come risultato della conversione: per tale motivo è porta di ingresso ed è collegata ai piedini dell'ADC0809.
Per quanto concerne la metà della porta C che è utilizzata come porta di uscita, le linee C0, C1, C2 sono utilizzate per indicare al convertitore su quale degli otto canali di ingresso analogici esso deve effettuare la conversione.
La linea C3 è utilizzata per un duplice scopo e precisamente:

1. indica che l'indirizzo presente su C0, C1 e C2 è quello del canale su cui dovrà essere effettuata la conversione;
2. fa iniziare la conversione stessa.

Il bit C4 che è, a differenza dei precedenti e assieme a C5, C6 e C7, di ingresso per la porta C, serve invece per testare il segnale di fine conversione (EOC) generato dall'ADC0809 quando ha terminato appunto una conversione sul canale precedentemente impostogli.
Il test su C4 non è necessario se il programma di acquisizione dei dati è scritto in Basic dato che tra il comando di inizio di conversione e l'istante in cui si effettua una lettura del dato convertito trascorre un tempo più che sufficiente perché la conversione stessa sia stata correttamente terminata, data la relativa "lentezza" del Basic.
Se invece il programma di gestione dell'ADC è scritto in linguaggio macchina è necessario testare il segnale EOC per essere sicuri di acquisire un dato corretto.

IL CONVERTITORE ADC0809

Questo dispositivo realizza una conversione analogico-digitale su otto canali.
Tramite segnali imposti agli ingressi di indirizzo ADD0-ADD2 è possibile scegliere il canale analogico su cui effettuare la conversione stessa.
Questa è di tipo raziometrico: ciò vuol dire che il risultato della conversione è un numero, un byte, di valore compreso tra 0 e 255, che rappresenta il rapporto tra il valore di tensione presente all'ingresso analogico e il valore di tensione applicato tra gli ingressi di riferimento VR+ e VR−.
Vediamo con un esempio cosa questo significa: per ipotesi sia imposta al piedino VR+ una tensione di +3 volt rispetto al piedino VR− e quest'ultimo sia collegato direttamente alla tensione di riferimento della tensione di alimentazione.

Se la tensione presente all'ingresso di un canale è minore o uguale a 0 volt il valore convertito è 0, se invece è maggiore o uguale di 3 volt tale valore è di 255.
Per i valori intermedi il valore fornito dal convertitore si ricava dalla:

$$valore = ((Vin - VR-)/(VR+ - VR-)) \times 255 = Vin \times 255 / VR+$$

dato che VR− = 0 volt. (Per ulteriori dettagli si rimanda all'appendice dove sono descritte le caratteristiche tecniche dell'ADC0809).

IL SOFTWARE

La routine per l'utilizzo del convertitore si articola in due parti distinte: la prima riguarda la programmazione dell'8255A, la seconda l'acquisizione dei dati convertiti.

```
 10 PA = 38912: PB=38913: PC=38914: CTL=38915
 20 POKE CLT,138
 30 REM FINE PROGRAMMAZIONE
 40 REM INIZIO CONVERSIONE
 50 FOR I = 0 TO 7
 60 POKE PC, I : REM INDIVIDUA L'INGRESSO ANALOGICO
 70 POKE CNT, 7 : REM GENERAZIONE DELL'IMPULSO DI START
    OF CONVERSION
 80 POKE CNT, 6
 90 A(I) = PEEK(B): REM ACQUISIZIONE DEL DATO CONVER-
    TITO
100 NEXT I : RETURN
```

Al termine di questa brevissima subroutine nel vettore A(I) sono presenti i risultati della conversione sull'ingresso *i*-esimo.
Lo stesso programma in linguaggio macchina è invece:

```
$1D00 LDA #$ 8A
$1D02 STA $ 9803
$1D05 RTS
$1D06 LDA $ 0341
$1D09 STA $ 9802
$1D0C LDA #$ 07
$1D0E STA $ 9803
$1D11 LDA #$ 06
```

```
$1D13 STA $ 9803
$1D16 LDA $ 9802
$1D19 AND #$ 10
$1D1B BNE $ 1D16
$1D1D LDA $ 9801
$1D20 STA $ 0341
$1D22 RTS
```

e può essere utilizzato dal Basic con le istruzioni:

```
SYS 29 * 256: REM PROGRAMMAZIONE DELL'8255A
POKE 833, NC: REM NUMERO DELL'INGRESSO ANALOGICO
SYS 29 * 256 + 6
A = PEEK(833): REM LETTURA DEL DATO
```

CONVERTITORE DI TEMPERATURA

Come esempio di utilizzazione del convertitore appena visto si descrive un trasduttore di temperatura. Lo schema è dato in figura 9.2.

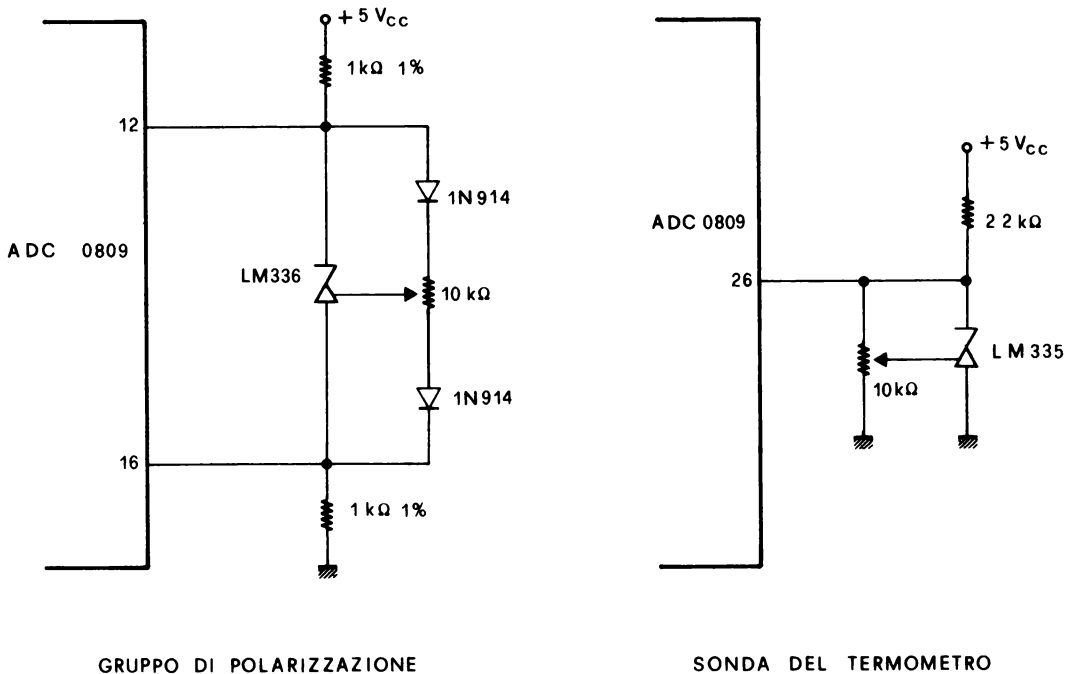



Fig. 9.2 Schema di un trasduttore di temperatura.

Il dispositivo LM335 è un trasduttore di temperatura a semiconduttore: polarizzato opportunamente al suo anodo è presente un valore di tensione legato alla temperatura dalla relazione:

V = 2,730 + 0,010 × °C

ove °C è la temperatura in gradi centigradi. La tensione è quindi proporzionale alla temperatura con un coefficiente di proporzionalità di 10 millivolt per grado centigrado.
La rete che genera la tensione da applicare agli ingressi di riferimento VR+ e VR− dell'ADC0809 si è resa necessaria allo scopo di avere una risoluzione decente nella conversione. Infatti se si fosse posto VR+ = 5 volt e VR− = 0 volt il convertitore avrebbe una capacità di discriminazione della tensione da convertire di:

5 / 256 = 0,019 volt

e quindi darebbe una variazione del valore convertito solo se la temperatura dell'LM335 variasse di circa 2 °C; avendo imposto ora a VR+ una tensione di 3,75 volt e a VR− di 1,25 volt, cosicché VR+ − VR− = 2,5 volt, si ha una risoluzione di 2,5 / 256 = 0,09 volt sufficiente a rilevare una variazione di temperatura di circa 1 °C.
Che letture ci possiamo aspettare dal convertitore? Facciamo qualche semplice calcolo:

Temperatura = 0 °C
V 335 = 2,73 volt

valore di uscita dal convertitore
= ((2,73 − 1,25) / 2,50) × 255 = 149

Temperatura = 10 °C
V 335 = 2,73 + 0,10 = 2,74 volt

valore di uscita dal convertitore
= ((2,74 − 1,25) / 2,50) × 255 = 161

Allora se vogliamo che nel nostro programma il vettore A(I) contenga il valore della temperatura in °C basta far eseguire l'istruzione:

```
A%(I) = A(I) × 10 / 12 − 1490/12
```

CAPITOLO DIECI

Un programmatore di EPROM

LE EPROM

Le *Electrically Programmable Read Only Memories* sono memorie a semiconduttore che nel loro normale modo di funzionamento assomigliano alle ROM.
Diversamente da queste è però possibile sia scrivere dati (byte) in qualsiasi loro locazione, cioè programmarle, per mezzo di opportuni impulsi di tensione, sia cancellarli utilizzando una sorgente di radiazione ultravioletta.
Una volta programmata, una EPROM conserva l'informazione anche se viene a mancare la tensione di alimentazione e in questo è identica a una ROM.
Esistono vari tipi di EPROM realizzati con tecnologie diverse; per quello che riguarda la loro capacità, in commercio se ne trovano comunemente da 2048 fino a 16536 locazioni.
Qualsiasi sia il tipo, le caratteristiche di lettura sono quelle di una normale memoria ROM o RAM: fornendo l'indirizzo della locazione che si vuole leggere e attivando l'ingresso di controllo per la lettura, esse presentano sul bus dei dati il contenuto della locazione indirizzata.
Per quanto riguarda la cancellatura occorre far notare che essa interessa tutte le locazioni presenti nella EPROM: dopo che questa ha assorbito della radiazione ultravioletta, in quantità specificata dal costruttore, tutte le locazioni presentano lo stesso valore, $00 o $FF a seconda del tipo.

La programmazione di una EPROM si attua fornendo agli ingressi di dati il valore che si vuole scrivere e agli ingressi di indirizzo quello relativo alla locazione interessata; una volta fatto questo occorre pilotare gli ingressi dedicati alla programmazione fornendo loro sia le tensioni che un impulso opportuno di programmazione, quest'ultimo di durata dell'ordine delle decine di millesimi di secondo.
È da notare inoltre che sia il dato che l'indirizzo della locazione in cui esso deve essere memorizzato debbono essere applicati alla EPROM per tutta la durata dell'impulso di programmazione.

SCHEMA DEL PROGRAMMATORE

Da quanto appena detto occorre costruire un hardware che presenti le seguenti caratteristiche:

1. deve fornire agli ingressi della EPROM sia l'indirizzo che il dato e mantenerli per tutto il tempo necessario a programmare una locazione;
2. deve gestire gli impulsi e la tensione di programmazione;
3. deve inoltre permettere di leggere il contenuto della locazione sulla quale si è appena "scritto" per verificare il buon esito della programmazione.

I primi due punti richiedono la realizzazione di una interfaccia.
Il 6502 infatti fornisce sì gli indirizzi e i dati sui bus omonimi per una operazione di scrittura ma li mantiene per intervalli di tempo dell'ordine dei microsecondi, non certamente per millisecondi; allora è necessario memorizzare nell'interfaccia sia il valore dell'indirizzo che quello del dato: solo facendo in questo modo si è sicuri che essi permangono per tutto il tempo necessario.
Per quanto riguarda il secondo punto la necessità dell'interfaccia è evidente dato che i valori delle tensioni di programmazione sono in genere più elevati di quelli presenti nel sistema a microprocessore.
La memorizzazione dei valori di indirizzo e di dato può avvenire molto semplicemente scrivendoli in un certo numero di porte di uscita; anche per i comandi relativi alla tensione e all'impulso di programmazione si sfruttano linee delle stesse porte.
Lo schema elettrico del programmatore è indicato in figura 10.1.
Come si può notare per l'interfaccia si è utilizzata anche qui la 8255A; le sue 24 linee di ingresso/uscita sono in numero sufficiente per programmare EPROM di tipo 2732A.
Queste contengono 4096 locazioni da un byte ciascuna: sono necessari

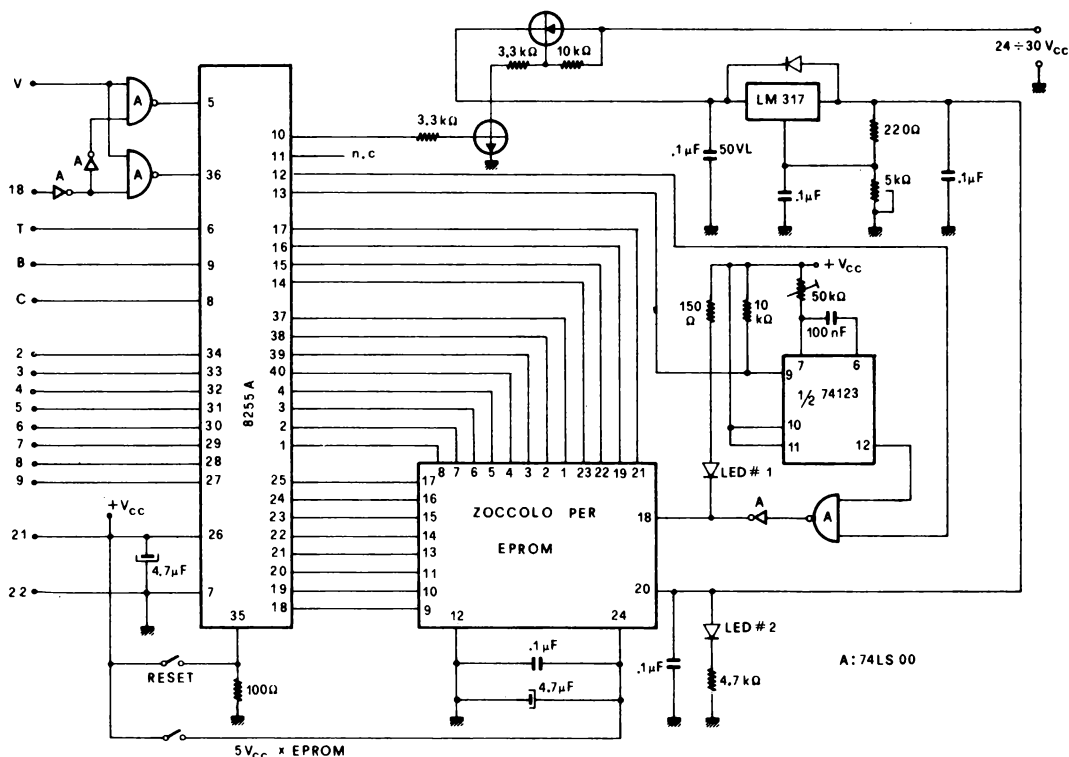



Fig. 10.1 Schema elettrico di un programmatore di EPROM.

allora 12 bit per l'indirizzo e 8 bit per il dato, pertanto restano a disposizione 4 linee di I/O tra quelle fornite dall'8255A.
Di queste una sarà utilizzata per comandare il generatore della tensione di programmazione (21 volt), una per la generazione del relativo impulso e una per l'abilitazione della EPROM stessa.
Per verificare la correttezza dei dati programmati occorre che le linee dati gestite dall'8255A siano bidirezionali, dato che i dati possono essere generati dall'8255A stesso (per la programmazione) o dal 2732A per la verifica.
Ciò è facilmente ottenibile con l'8255A essendo questo programmabile: allora si deve fare in modo che per la scrittura sulla EPROM una porta sia utilizzata come porta di uscita mentre per la verifica la medesima porta sia utilizzata come ingresso.
Si è usata come porta per i dati la porta B (ma si poteva scegliere in alternativa la A); gli indirizzi sono memorizzati nella porta A gli 8 bit meno significativi, in metà della porta C quelli più significativi.
Le caratteristiche del 2732A, riportate in figura 10.2, impongono per la programmazione di una locazione che:

A.C. PROGRAMMING CHARACTERISTICS (T_A = 25 ± 5°C, V_{CC} = 5V ± 5%, V_{PP} = 21V ± 0.5V)

Symbol	Parameter	Limits			Units	Test Conditions†
		Min.	Typ.	Max.		
t_{AS}	Address Setup Time	2			μs	
t_{OES}	$\overline{OE}$ Setup Time	2			μs	
t_{DS}	Data Setup Time	2			μs	
t_{AH}	Address Hold Time	0			μs	
t_{OEH}	OE Hold Time	2			μs	
t_{DH}	Data Hold Time	2			μs	
t_{DF}	Chip Enable to Output Float Delay	0		130	ns	
t_{DV}	Data Valid from $\overline{CE}$			1	μs	$\overline{CE} = V_{IL}$, $\overline{OE} = V_{IL}$
t_{PW}	$\overline{CE}$ Pulse Width During Programming	45	50	55	ms	
t_{PRT}	$\overline{OE}$ Pulse Rise Time During Programming	50			ns	
t_{VR}	V_{PP} Recovery Time	2			μs	

†A.C. TEST CONDITIONS

Input Rise and Fall Times (10% to 90%) .... 20 ns
Input Pulse Levels .... 0.45V to 2.4V
Input Timing Reference Level .... 1.0 and 2.0V
Output Timing Reference Level .... 0.8 and 2.0V

WAVEFORM

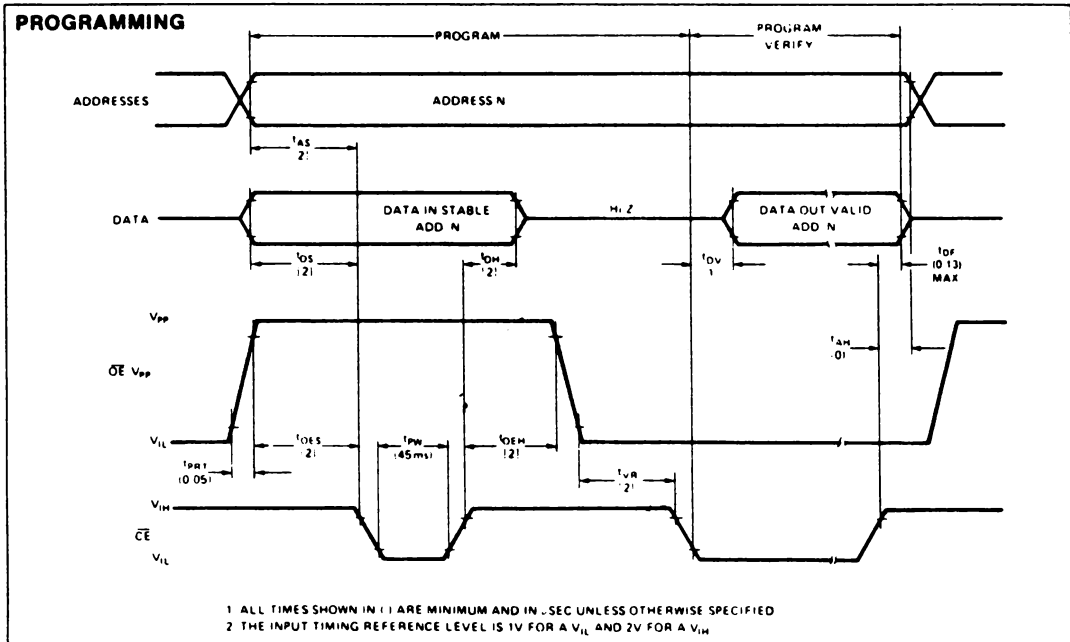



Fig. 10.2 Componenti data INTEL.

1. sia portato a livello logico alto l'ingresso /CE;
2. siano forniti i valori di indirizzo e di dato rispettivamente agli ingressi A11-A0 e O7-O0;
3. sia applicata la tensione di programmazione di 21 volt all'ingresso OE/Vpp;
4. sia fornito un impulso a livello logico basso all'ingresso /CE per un intervallo di tempo di 50 millisecondi (+− 10%);
5. sia tolta la tensione di programmazione.

Il punto 4) permette due scelte in alternativa: l'impulso può essere ottenuto sia per via software che per via hardware.
Si è scelta qui la seconda soluzione perché le specifiche di durata dell'impulso sono abbastanza strette e sarebbe stato abbastanza difficoltoso rispettarle in un programma scritto in Basic.
Come il lettore può rilevare dalla figura 10.1, il circuito è composto da tre sezioni principali: l'8255A assieme a un 74LS00 per l'interfaccia con i bus del calcolatore, il generatore di impulso e l'alimentatore, comandato dall'8255A, che fornisce la tensione di 21 volt per la programmazione.
Per quanto riguarda la prima sezione essa è identica a quella già vista nel caso del convertitore analogico digitale; l'unica differenza è costituita dal pulsante di reset il quale deve essere attivato per un istante, dopo che è stato acceso il calcolatore, per predisporre l'8255A in uno stato iniziale corretto.
La seconda sezione, composta dal monostabile e da alcune porte di tipo NAND e NOT serve per due scopi distinti: essa fornisce infatti a seconda dei segnali provenienti dai piedini 12 e 13 dell'8255A, rispettivamente il segnale /CE per la lettura del contenuto di una locazione della EPROM, a scopo di verifica, oppure l'impulso di programmazione.
Quest'ultimo è generato dal monostabile 74123, su comando proveniente dal piedino 13; si noti che lo scatto del monostabile avviene sul fronte negativo del comando. Il potenziometro collegato al piedino 7 del 74123 permette di regolare a 50 millisecondi esatti la durata dell'impulso.
Il diodo LED associato a questa sezione permette di vedere se sono in corso attività che coinvolgono il 2732A.
Per quanto riguarda l'alimentatore comandato, gli si deve fornire dall'esterno una tensione di 25-30 volt in continua; un segnale a valore logico alto proveniente dal piedino 10 dell'8255A lo mette in funzione per fornire la tensione di programmazione. Questa deve essere di 21 + −0,5 volt e la si può regolare al valore esatto tramite il potenziometro di 5 kohm.
Infine l'interruttore tra la tensione di +5 volt e il piedino 24 del 2732A permette di alimentarla quando il programma dà l'avviso.

Per quanto riguarda l'effettivo uso della scheda di programmazione e del relativo programma si debbono osservare alcune avvertenze. Il circuito del programmatore deve essere montato su una scheda che abbia un connettore compatibile a quello del bus di espansione del VIC-20; per l'inserimento della scheda valgono le usuali avvertenze cioè deve avvenire a calcolatore spento.

```
        PROGRAMMA PER LA SCHEDA EPROM

10 S1 = 36875 : V = 36878 : PRINT CHR$(147): GOSUB 750:GOSUB 370
20 ZZ$ = "//////"
30 REM****************
31 REM*INDIRIZZI DI *
32 REM* CONTROLLO   *
33 REM****************
34 :
40 PA = 38912 : PB = PA + 1 : PC = PB + 1 : CT = PC + 1
50 PH = 9 : PL = PH-1 : VH=15 :VL=VH-1 :CH=11 :CL=CH-1 :BI=130
:BO=128
51 :
52 :
55 REM****************
56 REM*TUTTE LE PORTE
57 REM* = INGRESSI
58 REM****************
60 POKE CT,BI:POKE PC,0:POKE PA,0
61 :
70 PRINT "VERIFICA SE I LED #1 E #2 SONO ACCESI": GOSUB740
75 :
80 PRINT "PRINT OK PER CONTINUARE"
90 INPUT G$: IF G$ ="OK" THEN 110
100 GOSUB 740: PRINT"/END/": END
110 PRINT "INSERISCI LA EPROM ":GOSUB 370
120 PRINT "APPLICA I 5 VOLT": GOSUB 370
125 REM****************
126 REM*VERIFICA DELLA*
127 REM*   EPROM      *
128 REM****************
129 :
130 PRINT "STO VERIFICANDO LA EPROM"
140 FOR I= 0 TO 4095
150 PRINT "." ;
160 GOSUB 450
170 :
180 W%=PEEK(PB): IF W% <> 255 THEN PRINT W%,I:POKE
PC,0:POKEPA,0:GOTO360
190 NEXT: PRINT"EPROM E' OK" : GOSUB370
195 :
200 PRINT "STO PROGRAMMANDO": GOSUB 370
205 REM****************
206 REM* PB IN USCITA*
207 REM* CE    ALTO  *
208 REM* PH    ALTO  *
209 :
210 POKE CT , BO: POKE PC,0 :POKE CT,PH:POKE CT,CH
211 :
220 PRINT "VERIFICA CHE SOLO UN LED E' ACCESO":GOSUB370
230 POKE CT,PH :PRINT" APPLICA I 5 VOLT": GOSUB370
240 PRINT "APPLICA I 24 VOLT": GOSUB370
244 REM****************
245 REM*RICHIESTA DEL*
```

```
246 REM*NOME DEL FILE*
247 REM*APERTURA DI   *
248 REM*CANALE        *
249 :
250 PRINT"IL NOME DEL FILE?": GOSUB740: INPUT K$
260 OPEN 1,1,0,K$
270 FOR I = 0 TO 4095: POKE CT,BO
280 PRINT I
290 INPUT# 1 , W%
300 GOSUB 580: REM ROUTINE DI PROGRAMMAZIONE
310 NEXT I
311 :
320 CLOSE 1
321 REM***************
322 REM* FINE PROGRAM-
323 REM* MAZIONE      *
324 :
325 REM***************
326 REM*TUTTE LE PORTE
327 REM* = INGRESSO
328 :
330 POKE PA,0: POKE PC,0: POKE CT,BI
340 GOSUB 740
350 PRINT "TOGLI I 24 VOLT": GOSUB370
360 GOSUB 740: PRINT "TOGLI I 5 VOLT": GOSUB740: PRINT"TOGLI LA
EPROM":GOTO 100
365 REM***************
366 REM* TEST PER     *
367 REM* CONTINUARE   *
368 :
370 GOSUB 740:PRINT"PREMI C PER CONTINUARE","N = STOP","H = MENU"
380 POKE 198,0
390 GET G$
400 IF G$="C" THEN 440
410 IF G$="H" THEN GOSUB750
420 IF G$="N" THEN POKE CT,BI: GOTO  360
430 GOTO390
440 RETURN
441 :
442 :
443 REM***************
444 REM*CALCOLO DELLA*
445 REM*LOCAZIONE  E *
446 REM*INVIO IN PA E*
447 REM* IN PC       *
448 REM***************
450 AH%=INT(I/256):AL%=I-256*AH%:POKEPA,AL%:POKEPC,(PEEK(PC)AND
240)OR AH%:RETURN
451 :
452 :
455 REM***************
456 REM*CONFRONTO DELLA
457 REM*EPROM CON IL *
458 REM*FILE SU NASTRO
459 :
460 POKE CT,BI: POKE PC,0:POKE PA,0
470 PRINT"IL NOME DEL FILE? ":GOSUB740: INPUT K$
480 OPEN 1,1,0,K$
490 FORI=0 TO 4095
500 PRINT I
510 POKE CT,BI
520 GOSUB 450
530 INPUT# 1,W%
540 R% =PEEK(PB): PRINTW%,R%;:IF R% <> W% THEN PRINT"**ERRORE**":STOP
550 NEXT
```

```
560 GOTO 320
570 :
571 REM****************
572 REM*ROUTINE  DI  *
573 REM*PROGRAMMAZIONE
574 REM****************
575 :
580 NN = 0:F = 0
585 REM****************
586 REM*SE IL BYTE DA*
587 REM*SCRIVERE E'  *
588 REM*IDENTICO:RETURN
589 :
590 POKE CT,BI: GOSUB 450: IF W% = PEEK(PB) THEN RETURN
595 REM****************
596 REM* PB IN USCITA*
597 REM* 24 VOLT : ON*
598 :
600 POKE CT,BO: POKE CT,PH: POKECT,CH: POKE CT,VH
610 GOSUB 450
615 REM****************
616 REM*SCRITTURA IN PB
617 :
620 POKE PB,W%
615 REM****************
616 REM*LOOP DI       *
617 REM*PROGRAMMAZIONE
618 :
630 FOR XX= 0TO1: POKE CT,PL:POKE CT,PH:FOR TT=1TO 5 :NEXT TT,XX
635 REM****************
636 REM*24 VOLT : OFF*
637 :
640 POKE CT,VL
645 REM****************
646 REM*LOOP DI ATTESA
647 :
650 FOR TT=1TO 1 :NEXT
651 :
655 REM****************
656 REM*  VERIFICA    *
657 :
660 POKE CT,BI
670 GOSUB 450
680 R% = PEEK(PB): PRINT I ; W% , R%;
690 IF NN = 3 THEN 320
700 IF R% <> W% THEN NN = NN +1: F=0: GOTO600
710 IF F=1 THEN RETURN
720 IF R% = W% THEN F=1:GOTO 600
730 RETURN
735 :
740 REM***************
741 REM* BUZZER      *
742 REM***************
743 :
745 POKE S1,200:POKE V,15:FOR TS=1TO200: NEXT:POKE V,0:RETURN
746 :
747 REM***************
748 REM*# DI ISTRUZ.*
749 REM* PER GOTO    *
750 PRINT "200=PROGRAMMAZIONE" ,"330=FINE" ,"470=VERIFICA NASTRO":
RETURN
```

Una volta che la scheda sia inserita si accende il calcolatore e si dà il reset all'8255A per i motivi detti precedentemente; la EPROM da programmare non deve essere inserita nello zoccolo e l'interruttore dei +5 volt deve essere aperto. Si fornisce poi la tensione all'alimentatore comandato e si carica il programma: seguendo le indicazioni fornite da quest'ultimo si inserirà la EPROM e si agirà sui vari interruttori.
Si noti che il programma preleva i dati da memorizzare sul 2732A da un file su nastro o su disco sul quale i dati sono stati scritti utilizzando la istruzione:

```
PRINT # X, DATO
```

dove 0 ≤ DATO ≤= 255; si può evidentemente migliorare utilizzando invece un file di byte con il quale il file, a parità di dati, risulta molto più corto.
Quale può essere l'utilizzazione del programmatore è presto detto: se si vogliono avere a disposizione routine in linguaggio macchina da noi create, tipicamente dei cunei nell'interprete Basic o addirittura nel KERNAL, conviene averle su EPROM da inserire nel bus di espansione; un altro utilizzo è di avere una copia delle cartucce ROM per uso personale.

APPENDICE 1

Codici del Basic del VIC-20

Codice decimale	Caratteri/ tasti	Codice decimale	Caratteri/ tasti	Codice decimale	Caratteri/ tasti	Codice decimale	Caratteri/ tasti
0	End of line	66	B	133	INPUT	169	STEP
1–31	Unused	67	C	134	DIM	170	+
32	space	68	D	135	READ	171	–
33	!	69	E	136	LET	172	•
34	”	70	F	137	GOTO	173	/
35	#	71	G	138	RUN	174	|
36	$	72	H	139	IF	175	AND
37	%	73	I	140	RESTORE	176	OR
38	&	74	J	141	GOSUB	177	>
39	’	75	K	142	RETURN	178	=
40	(	76	L	143	REM	179	<
41	)	77	M	144	STOP	180	SGN
42	*	78	N	145	ON	181	INT
43	+	79	O	146	WAIT	182	ABS
44	,	80	P	147	LOAD	183	USR
45	–	81	Q	148	SAVE	184	FRE
46	•	82	R	149	VERIFY	185	POS
47	/	83	S	150	DEF	186	SQR
48	0	84	T	151	POKE	187	RND
49	1	85	U	152	PRINT#	188	LOG
50	2	86	V	153	PRINT	189	EXP
51	3	87	W	154	CONT	190	COS
52	4	88	X	155	LIST	191	SIN
53	5	89	Y	156	CLR	192	TAN
54	6	90	Z	157	CMD	193	ATN
55	7	91	[	158	SYS	194	PEEK
56	8	92	\	159	OPEN	195	LEN
57	9	93	]	160	CLOSE	196	STR$
58	:	94	↑	161	GET	197	VAL
59	;	95	←	162	NEW	198	ASC
60	<	96–127	Unused	163	TAB(	199	CHR$
61	=	128	END	164	TO	200	LEFT$
62	>	129	FOR	165	FN	201	RIGHT$
63	?	130	NEXT	166	SPC(	202	MID$
64	@	131	DATA	167	THEN	203–254	Unused
65	A	132	INPUT#	168	NOT	255	π

Istruzioni del 6502

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

ADC *Add Memory to Accumulator with Carry* ADC

Operation: A + M + C → A, C

N Z C I D V
✓ ✓ ✓ — — ✓

(Ref: 2.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ADC # Oper	69	2	2
Zero Page	ADC Oper	65	2	3
Zero Page, X	ADC Oper, X	75	2	4
Absolute	ADC Oper	6D	3	4
Absolute, X	ADC Oper, X	7D	3	4*
Absolute, Y	ADC Oper, Y	79	3	4*
(Indirect, X)	ADC (Oper, X)	61	2	6
(Indirect), Y	ADC (Oper), Y	71	2	5*

* Add 1 if page boundary is crossed.

AND *"AND" Memory with Accumulator* AND

Logical AND to the accumulator

Operation: A ∧ M → A

N Z C I D V
✓ ✓ — — — —

(Ref: 2.2.3.0)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	AND # Oper	29	2	2
Zero Page	AND Oper	25	2	3
Zero Page, X	AND Oper, X	35	2	4
Absolute	AND Oper	2D	3	4
Absolute, X	AND Oper, X	3D	3	4*
Absolute, Y	AND Oper, Y	39	3	4*
(Indirect, X)	AND (Oper, X)	21	2	6
(Indirect), Y	AND (Oper), Y	31	2	5

* Add 1 if page boundary is crossed.

ASL ASL *Shift Left One Bit (Memory or Accumulator)* ASL

Operation: C ← [7|6|5|4|3|2|1|Ø] ← Ø

N Z C I D V
✓ ✓ ✓ — — —

(Ref: 10.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	ØA	1	2
Zero Page	ASL Oper	Ø6	2	5
Zero Page, X	ASL Oper, X	16	2	6
Absolute	ASL Oper	ØE	3	6
Absolute, X	ASL Oper, X	1E	3	7

BCC BCC *Branch on Carry Clear* BCC

Operation: Branch on C = Ø

N Z C I D V
— — — — — —

(Ref: 4.1.1.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC Oper	9Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BCS BCS *Branch on Carry Set* BCS

Operation: Branch on C = 1

N Z C I D V
— — — — — —

(Ref: 4.1.1.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS Oper	BØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BEQ BEQ *Branch on Result Zero* BEQ

Operation: Branch on Z = 1

N Z C I D V
— — — — — —

(Ref: 4.1.1.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ Oper	FØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BIT BIT *Test Bits in Memory with Accumulator* BIT

Operation: A ∧ M, M_7 → N, M_6 → V

Bit 6 and 7 are transferred to the status register. If the result of A∧M is zero then Z = 1, otherwise Z = Ø

N Z C I D V
M_7 ✓ — — — M_6

(Ref: 4.2.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT Oper	24	2	3
Absolute	BIT Oper	2C	3	4

BMI BMI *Branch on Result Minus* BMI

Operation: Branch on N = 1

N Z C I D V
— — — — — —

(Ref: 4.1.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI Oper	3Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BNE BNE *Branch on Result not Zero* BNE

Operation: Branch on Z = 0

N Z C I D V
— — — — — —

(Ref: 4.1.1.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE Oper	DØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BPL — BPL *Branch on Result Plus* — BPL

Operation: Branch on N = 0   N Z C I D V: — — — — — —

(Ref: 4.1.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL Oper	10	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BRK — BRK *Force Break* — BRK

Operation: Forced Interrupt PC + 2 ↓ P ↓   N Z C I D V: — — — 1 — —

(Ref: 9.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	00	1	7

1. A BRK command cannot be masked by setting I.

BVC — BVC *Branch on Overflow Clear* — BVC

Operation: Branch on V = 0   N Z C I D V: — — — — — —

(Ref: 4.1.1.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC Oper	50	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BVS — BVS *Branch on Overflow Set* — BVS

Operation: Branch on V = 1   N Z C I D V: — — — — — —

(Ref: 4.1.1.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS Oper	70	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

CLC — CLC *Clear Carry Flag* — CLC

Operation: 0 → C   N Z C I D V: — — 0 — — —

(Ref: 3.0.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLC	18	1	2

CLD — CLD *Clear Decimal Mode* — CLD

Operation: 0 → D   N Z C I D V: — — — — 0 —

(Ref: 3.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI — CLI *Clear Interrupt Disable Bit* — CLI

Operation: 0 → I   N Z C I D V: — — — 0 — —

(Ref: 3.2.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV — CLV *Clear Overflow Flag* — CLV

Operation: 0 → V   N Z C I D V: — — — — — 0

(Ref: 3.6.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP — CMP *Compare Memory and Accumulator* — CMP

Operation: A − M   N Z C I D V: ✓ ✓ ✓ — — —

(Ref: 4.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CMP #Oper	C9	2	2
Zero Page	CMP Oper	C5	2	3
Zero Page, X	CMP Oper, X	D5	2	4
Absolute	CMP Oper	CD	3	4
Absolute, X	CMP Oper, X	DD	3	4*
Absolute, Y	CMP Oper, Y	D9	3	4*
(Indirect, X)	CMP (Oper, X)	C1	2	6
(Indirect), Y	CMP (Oper), Y	D1	2	5*

* Add 1 if page boundary is crossed.

CPX — CPX *Compare Memory and Index X* — CPX

Operation X − M   N Z C I D V: ✓ ✓ ✓ — — —

(Ref: 7.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPX #Oper	E0	2	2
Zero Page	CPX Oper	E4	2	3
Absolute	CPX Oper	EC	3	4

CPY — CPY *Compare Memory and Index Y* — CPY

Operation: Y - M

N Z C I D V
✓ ✓ ✓ – – –

(Ref: 7.9)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPY #Oper	C0	2	2
Zero Page	CPY Oper	C4	2	3
Absolute	CPY Oper	CC	3	4

DEC — DEC *Decrement Memory by One* — DEC

Operation: M - 1 → M

N Z C I D V
✓ ✓ – – – –

(Ref: 10.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC Oper	C6	2	5
Zero Page, X	DEC Oper, X	D6	2	6
Absolute	DEC Oper	CE	3	6
Absolute, X	DEC Oper, X	DE	3	7

DEX — DEX *Decrement Index X by One* — DEX

Operation: X - 1 → X

N Z C I D V
✓ ✓ – – – –

(Ref: 7.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY — DEY *Decrement Index Y by One* — DEY

Operation: Y - 1 → Y

N Z C I D V
✓ ✓ – – – –

(Ref: 7.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR — EOR *"Exclusive-Or" Memory with Accumulator* — EOR

Operation: A ⊻ M → A

N Z C I D V
✓ ✓ – – – –

(Ref: 2.2.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	EOR #Oper	49	2	2
Zero Page	EOR Oper	45	2	3
Zero Page, X	EOR Oper, X	55	2	4
Absolute	EOR Oper	4D	3	4
Absolute, X	EOR Oper, X	5D	3	4*
Absolute, Y	EOR Oper, Y	59	3	4*
(Indirect, X)	EOR (Oper, X)	41	2	6
(Indirect),Y	EOR (Oper), Y	51	2	5*

* Add 1 if page boundary is crossed.

INC — INC *Increment Memory by One* — INC

Operation: M + 1 → M

N Z C I D V
✓ ✓ – – – –

(Ref: 10.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	INC Oper	E6	2	5
Zero Page, X	INC Oper, X	F6	2	6
Absolute	INC Oper	EE	3	6
Absolute, X	INC Oper, X	FE	3	7

INX — INX *Increment Index X by One* — INX

Operation: X + 1 → X

N Z C I D V
✓ ✓ – – – –

(Ref: 7.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY — INY *Increment Index Y by One* — INY

Operation: Y + 1 → Y

N Z C I D V
✓ ✓ – – – –

(Ref: 7.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP — JMP *Jump to New Location* — JMP

Operation: (PC + 1) → PCL
(PC + 2) → PCH

(Ref: 4.0.2)
(Ref: 9.8.1)

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JMP Oper	4C	3	3
Indirect	JMP (Oper)	6C	3	5

JSR — JSR *Jump to New Location Saving Return Address* — JSR

Operation: PC + 2 ↓, (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

(Ref: 8.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR Oper	20	3	6

LDA — LDA *Load Accumulator with Memory* — LDA

Operation: M → A   N Ƶ C I D V
(Ref: 2.1.1)   ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDA #Oper	A9	2	2
Zero Page	LDA Oper	A5	2	3
Zero Page, X	LDA Oper, X	B5	2	4
Absolute	LDA Oper	AD	3	4
Absolute, X	LDA Oper, X	BD	3	4*
Absolute, Y	LDA Oper, Y	B9	3	4*
(Indirect, X)	LDA (Oper, X)	A1	2	6
(Indirect), Y	LDA (Oper), Y	B1	2	5*

* Add 1 if page boundary is crossed.

LDX — LDX *Load Index X with Memory* — LDX

Operation: M → X   N Ƶ C I D V
(Ref: 7.0)   ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDX # Oper	A2	2	2
Zero Page	LDX Oper	A6	2	3
Zero Page, Y	LDX Oper, Y	B6	2	4
Absolute	LDX Oper	AE	3	4
Absolute, Y	LDX Oper, Y	BE	3	4*

* Add 1 when page boundary is crossed.

LDY — LDY *Load Index Y with Memory* — LDY

Operation: M → Y   N Ƶ C I D V
(Ref: 7.1)   ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDY #Oper	AØ	2	2
Zero Page	LDY Oper	A4	2	3
Zero Page, X	LDY Oper, X	B4	2	4
Absolute	LDY Oper	AC	3	4
Absolute, X	LDY Oper, X	BC	3	4*

* Add 1 when page boundary is crossed.

LSR — LSR *Shift Right One Bit (Memory or Accumulator)* — LSR

Operation: Ø → [7|6|5|4|3|2|1|0] → C   N Ƶ C I D V
(Ref: 10.1)   Ø ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	LSR A	4A	1	2
Zero Page	LSR Oper	46	2	5
Zero Page, X	LSR Oper, X	56	2	6
Absolute	LSR Oper	4E	3	6
Absolute, X	LSR Oper, X	5E	3	7

NOP — NOP *No Operation* — NOP

Operation: No Operation (2 cycles)   N Ƶ C I D V
- - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA — ORA *"OR" Memory with Accumulator* — ORA

Operation: A V M → A   N Ƶ C I D V
(Ref: 2.2.3.1)   ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ORA #Oper	Ø9	2	2
Zero Page	ORA Oper	Ø5	2	3
Zero Page, X	ORA Oper, X	15	2	4
Absolute	ORA Oper	ØD	3	4
Absolute, X	ORA Oper, X	1D	3	4*
Absolute, Y	ORA Oper, Y	19	3	4*
(Indirect, X)	ORA (Oper, X)	Ø1	2	6
(Indirect), Y	ORA (Oper), Y	11	2	5

* Add 1 on page crossing

PHA — PHA *Push Accumulator on Stack* — PHA

Operation: A ↓   N Ƶ C I D V
(Ref: 8.5)   - - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP — PHP *Push Processor Status on Stack* — PHP

Operation: P↓   N Ƶ C I D V
(Ref: 8.11)   - - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	Ø8	1	3

PLA — PLA *Pull Accumulator from Stack* — PLA

Operation: A ↑   N Ƶ C I D V
(Ref: 8.6)   ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP — PLP *Pull Processor Status from Stack* — PLP

Operation: P ↑ — N Z C I D V: From Stack

(Ref: 8.12)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL — ROL *Rotate One Bit Left (Memory or Accumulator)* — ROL

Operation: M or A: 7 6 5 4 3 2 1 0 ← C ← — N Z C I D V: ✓ ✓ ✓ — — —

(Ref: 10.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROL A	2A	1	2
Zero Page	ROL Oper	26	2	5
Zero Page, X	ROL Oper, X	36	2	6
Absolute	ROL Oper	2E	3	6
Absolute, X	ROL Oper, X	3E	3	7

ROR — ROR *Rotate One Bit Right (Memory or Accumulator)* — ROR

Operation:

N Z C I D V: ✓ ✓ ✓ — — —

(Ref: 10.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR Oper	66	2	5
Zero Page,X	ROR Oper,X	76	2	6
Absolute	ROR Oper	6E	3	6
Absolute,X	ROR Oper,X	7E	3	7

Note: ROR instruction will be available on MCS650X microprocessors after June, 1976.

RTI — RTI *Return from Interrupt* — RTI

Operation: P↑ PC↑ — N Z C I D V: From Stack

(Ref: 9.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	40	1	6

RTS — RTS *Return from Subroutine* — RTS

Operation: PC↑, PC + 1 → PC — N Z C I D V: — — — — — —

(Ref: 8.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	60	1	6

SBC — SBC *Subtract Memory from Accumulator with Borrow* — SBC

Operation: A - M - $\bar{C}$ → A — N Z C I D V: ✓ ✓ ✓ — — ✓

Note: $\bar{C}$ = Borrow (Ref: 2.2.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	SBC #Oper	E9	2	2
Zero Page	SBC Oper	E5	2	3
Zero Page, X	SBC Oper, X	F5	2	4
Absolute	SBC Oper	ED	3	4
Absolute, X	SBC Oper, X	FD	3	4*
Absolute, Y	SBC Oper, Y	F9	3	4*
(Indirect, X)	SBC (Oper, X)	E1	2	6
(Indirect), Y	SBC (Oper), Y	F1	2	5*

* Add 1 when page boundary is crossed.

SEC — SEC *Set Carry Flag* — SEC

Operation: 1 → C — N Z C I D V: — — 1 — — —

(Ref: 3.0.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED — SED *Set Decimal Mode* — SED

Operation: 1 → D — N Z C I D V: — — — — 1 —

(Ref: 3.3.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI — SEI *Set Interrupt Disable Status* — SEI

Operation: 1 → I — N Z C I D V: — — — 1 — —

(Ref: 3.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA — STA *Store Accumulator in Memory* — STA

Operation: A → M — N Z C I D V: — — — — — —

(Ref: 2.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA Oper	85	2	3
Zero Page, X	STA Oper, X	95	2	4
Absolute	STA Oper	8D	3	4
Absolute, X	STA Oper, X	9D	3	5
Absolute, Y	STA Oper, Y	99	3	5
(Indirect, X)	STA (Oper, X)	81	2	6
(Indirect), Y	STA (Oper), Y	91	2	6

STX — STX *Store Index X in Memory* — STX

Operation: X → M

N Z C I D V
\- - - - - -

(Ref: 7.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX Oper	86	2	3
Zero Page, Y	STX Oper, Y	96	2	4
Absolute	STX Oper	8E	3	4

STY — STY *Store Index Y in Memory* — STY

Operation: Y → M

N Z C I D V
\- - - - - -

(Ref: 7.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY Oper	84	2	3
Zero Page, X	STY Oper, X	94	2	4
Absolute	STY Oper	8C	3	4

TAX — TAX *Transfer Accumulator to Index X* — TAX

Operation: A → X

N Z C I D V
✓ ✓ - - - -

(Ref: 7.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY — TAY *Transfer Accumulator to Index Y* — TAY

Operation: A → Y

N Z C I D V
✓ ✓ - - - -

(Ref: 7.13)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TSX — TSX *Transfer Stack Pointer to Index X* — TSX

Operation: S → X

N Z C I D V
✓ ✓ - - - -

(Ref: 8.9)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA — TXA *Transfer Index X to Accumulator* — TXA

Operation: X → A

N Z C I D V
✓ ✓ - - - -

(Ref: 7.12)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS — TXS *Transfer Index X to Stack Pointer* — TXS

Operation: X → S

N Z C I D V
\- - - - - -

(Ref: 8.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

TYA — TYA *Transfer Index Y to Accumulator* — TYA

Operation: Y → A

N Z C I D V
✓ ✓ - - - -

(Ref: 7.14)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

APPENDICE 3

Mappa di memoria del VIC-20 primo blocco da 1 K

Tabella 1 Mappa di memoria del VIC-20 del primo blocco da 1K

ESADEC.	DECIMALE	DESCRIZIONE
0000	0	Jump for USR
0001–0002	1–2	USR vector
0003–0004	3–4	Float–Fixed vector
0005–0006	5–6	Fixed–Float vector
0007	7	Search character, ':' or endline
0008	8	Scan between quotes flag
0009	9	TAB, column save position of cursor on line
000A	10	0 = LOAD, 1 = VERIFY
000B	11	Input, buffer pointer/#subscript
000C	12	Default DIM flag
000D	13	Type: FF = string, 00 = numeric
000E	14	Type: 80 = integer, 00 = floating point
000F	15	DATA scan/LIST quote/memory flag
0010	16	Subscript/FNx flag
0011	17	0 = INPUT; $40 = GET; $98 = READ
0012	18	ATN sign/Comparison eval flag
0013	19	Current I/O prompt flag
0014–0015	20–21	Basic integer address for SYS,GOTO etc.
0016	22	Pointer: temporary string stack
0017–0018	23–24	Last temp string vector
0019–0021	25–33	Stack for temporary strings
0022–0025	34–37	Utility pointer area

Tabella 1 *Continua*

ESADEC.	DECIMALE	DESCRIZIONE
0026–002A	38–42	Product area for multiplication
002B–002C	43–44	Pointer: Start of user BASIC (bottom of memory)
002D–002E	45–46	Pointer: Start of Variables
002F–0030	47–48	Pointer: Start of Arrays
0031–0032	49–50	Pointer: End of Arrays
0033–0034	51–52	Pointer: String storage (moving down)
0035–0036	53–54	Pointer: Top of active strings
0037–0038	55–56	Pointer: End of user BASIC (top of memory)
0039–003A	57–58	Current Basic line number
003B–003C	59–60	Previous Basic line number
003D–003E	61–62	Pointer: Basic statement for CONT
003F–0040	63–64	Current DATA line number
0041–0042	65–66	Current DATA item address
0043–0044	67–68	Input vector
0045–0046	69–70	Current variable name
0047–0048	71–72	Current variable address
0049–004A	73–74	Variable pointer for FOR/NEXT
004B–004C	75–76	Y-save; op-save; Basic pointer save
004D	77	Comparison symbol accumulator
004E	78–83	Misc. work area, Pointers, etc.
0054–0056	84–86	Jump vector for functions
0057–0060	87–96	Misc. numeric work area
0061	97	Accum#1: Exponent
0062–0065	98–101	Accum#1: Mantissa
0066	102	Accum#1: Sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi-order (overflow)
0069–006E	105–110	Accum#2: Exponent, Mantissa, etc
006F	111	Sign comparison, Acc#1 vs. Acc#2
0070	112	Accum#1 lo-order (rounding)
0071–0072	113–114	Cassette buffer length/Series pointer
0073–008A	115–138	CHARGET subroutine (Get Basic char)
007A–007B	122–123	Basic CHARGET vector (within subroutine)
008B–008F	139–143	RND seed value;
0090	144	Status word ST
00091	145	Keyswitch PIA: STOP (= $FE) and RVS flags
0092	146	Timing constant for tape
0093	147	Load = 0, Verify = 1
0094	148	Serial output: deferred char. flag (IEEE)
0095	149	Serial deferred character (IEEE)
0096	150	Tape EOT received
0097	151	Register save (IEEE)
0098	152	How many open files?
0099	153	Input device (normally 0)

Tabella 1 *Continua*

ESADEC.	DECIMALE	DESCRIZIONE
009A	154	Output (CMD) device (normally 3)
009B	155	Tape character parity
009C	156	Byte-received flag/cassette dipole switch
009D	157	OS message flag: Direct = $80/RUN = 0
009E	158	Tape Pass 1 error log/char. buffer
009F	159	Tape Pass 2 error log corrected
00A0–00A2	160–162	Jiffy Clock (HML)
00A3	163	Serial bit count/EOI flag
00A4	164	Cycle count for serial I/O
00A5	165	Countdown, tape write/bit count
00A6	166	Pointer: tape buffer
00A7	167	Tape write ldr count/Read pass/inhibit (RS232)
00A8	168	Tape Write new byte/Read error/inhibit cnt (RS232)
00A9	169	Write start bit/Read bit err/stbit (RS232)
00AA	170	Tape Scan;Ld;End/byte assy (RS232)
00AB	171	Write lead length/Rd checksum/parity (RS232)
00AC–00AD	172–173	Pointer: tape buffer, scrolling
00AE–00AF	174–175	Tape end addresses/End of program for SAVE
00B0–00B1	176–177	Tape timing constants
00B2–00B3	178–179	Pointer: start of tape buffer
00B4	180	Tape timer (1 = enable); bit cnt (RS232)
00B5	181	Tape EOT/next bit to send (RS232)
00B6	182	Read character error/outbyte buffer (RS232)
00B7	183	# characters in file name
00B8	184	Current logical file
00B9	185	Current secondary address or R/W
00BA	186	Current device
00BB–00BC	187–188	Pointer: to file name
00BD	189	Write shift word/Read input char (RS232)
00BE	190	# blocks remaining to Write/Read
00BF	191	Serial word buffer
00C0	192	Tape motor interlock
00C1–00C2	193–194	I/O start addresses
00C3–00C4	195–196	Kernal setup pointer
00C5	197	Current key pressed
00C6	198	# chars in keyboard buffer
00C7	199	Screen reverse flag (0 = off, 18 = on)
00C8	200	Pointer: End-of-line for input
00C9–00CA	201–202	Input cursor log (row, column)
00CB	203	Which key: 64 if no key
00CC	204	Cursor enable (0 = flash cursor on, 1 = off)
00CD	205	Cursor blink delay
00CE	206	Character under cursor
00CF	207	Cursor in blink phase flag (1 = off, 0 = visible)

Tabella 1 *Continua*

ESADEC.	DECIMALE	DESCRIZIONE
00D0	208	Input from screen/from keyboard
00D1–00D2	209–210	Pointer to screen line address
00D3	211	Position of cursor on above line
00D4	212	0 = direct cursor, else programmed
00D5	213	Current screen line length (22,44,66,88)
00D6	214	Row where cursor lives
00D7	215	Last inkey/checksum/buffer
00D8	216	# of INSERTs outstanding
00D9–00F0	217–240	Screen line link table
00F1	241	Dummy screen link
00F2	242	Screen row marker
00F3–00F4	243–244	Screen color printer
00F5–00F6	245–246	Keyboard pointer
00F7–00F8	247–248	Pointer RS-232 receive buffer base location
00F9–00FA	249–250	Pointer RS-232 transmit buffer base location
00FB–00FE	251–254	Operating system free zero page space
00FF	255	Basic storage
0100–010A	256–266	Floating to ASCII work area
0100–103E	256–318	Tape error log
0100–01FF	256–511	Processor stack area
0200–0258	512–600	BASIC input buffer
0259–0262	601–610	Logical file table
0263–026C	611–620	Device # table
026D–0276	621–630	Secondary address or R/W CMD table
0277–0280	631–640	Keyboard buffer
0281–0282	641–642	Start of memory for op system
0283–0284	643–644	Top of memory for op system
0285	645	Serial bus timeout flag (IEEE)
0286	646	Current color code
0287	647	Color under cursor
0288	648	Hi-byte base location of screen
0289	649	Max. size of keyboard buffer
028A	650	Key repeat 128=repeat all keys, 64=repeat no keys, 0=cursor controls
028B	651	Delay before first repeat occurs
028C	652	Delay between repeats
028D	653	Keyboard Shift/Control flag
028E	654	Last keyboard shift pattern
028F–0290	655–656	Pointer: keyboard decode table
0291	657	Shift mode switch (0=enabled, 128=locked)
0292	658	Auto scroll down flag (0 = on, 1 = off)
0293	659	Pseudo RS232 control register
0294	660	Pseudo RS232 command register
0295–0296	661–662	Non-standard bit time (2–100)

Tabella 1 *Continua*

ESADEC.	DECIMALE	DESCRIZIONE
0297	663	RS-232 status register
0298	664	Number of bits sent/received
0299–029A	665–666	Baud rate (full) bit time
029B	667	RS232 end of input buffer pointer
029C	668	RS232 start of input buffer pointer
029D	669	RS232 start of transmit buffer pointer
029E	670	RS232 end of transmit buffer pointer
029F–02A0	671–672	Holds IRQ during tape operations
02A1–02FF	673–767	Program indirects
0300–0301	768–769	Error message link
0302–0303	770–771	Basic warm start link
0304–0305	772–773	Crunch Basic tokens link
0306–0307	774–775	Print tokens link
0308–0309	776–777	Start new Basic code link
030A–030B	778–779	Get arithmetic element link
030C	780	Storage for 6502 .A register during SYS
030D	781	Storage for 6502 .X register during SYS
030E	782	Storage for 6502 .Y register during SYS
030F	783	Storage for 6502 .P register during SYS
0310–0313	784–787	??
0314–0315	788–789	IRQ interrupt vector (EABF)
0316–0317	790–791	BRK interrupt vector (FED2)
0318–0319	792–793	NMI interrupt vector (FEAD)
031A–031B	794–795	OPEN vector (F40A)
031C–031D	796–797	CLOSE vector (F34A)
031E–031F	798–799	Set-input vector (F2C7)
0320–0321	800–801	Set-output vector (F309)
0322–0323	802–803	Restore I/O vector (F3F3)
0324–0325	804–805	INPUT vector (F20E)
0326–0327	806–807	Output vector (F27A)
0328–0329	808–809	Test-STOP vector (F770)
032A–032B	810–811	GET vector (F1F5)
032C–032D	812–813	Abort I/O vector (F3EF)
032F–032F	814–815	User vector (FED2)
0330–0331	816–817	Link to load RAM (F549)
0332–0333	818–819	Link to save RAM (F685)
0334–033B	820–827	??
033C–03FB	828–1019	Cassette buffer

Tabella 2 Caratteri corrispondenti ai codici dello schermo e CBM ascii

	Schermo (POKE)	Schermo (PRINT)	Stampante (PRINT#)
Set 1	POKE 36869, PEEK(36869) AND 240	CHR$(142)	CHR$(145)
Set 2	POKE 36869, PEEK(36869) AND 242	CHR$(14)	CHR$(17)

Set 1 Maiuscole + Miscellanea/Grafica.
Set 2 Minuscole + Miscellanea/Maiuscole + Grafica.

	Schermo (PRINT)	Stampante (PRINT#)
Reverse field off	CHR$(146)	CHR$(146)
Reverse field on	CHR$(18)	CHR$(18)

Il codice ASCII (*American Standard Codefor Information Interchange*) è molto usato perla rappresentazione di caratteri. Normalmente è un codice di 7 bit che rappresenta 128 caratteri.
I computer CBM immagazzinano caratteri in una versione estesa di 8 bit del formato ASCII, consentendo una rappresentazione di 256 caratteri. In un testo in Basic 7 bit = 1 sta a significare una parola chiave; altrove nella memoria i codici di caratteri di 8 bit vengono interpretati come nella seguente tabella.

SET 1	SET 2	SCREEN CODE	CBM ASCII	SET 1	SET 2	SCREEN CODE	CBM ASCII	SET 1	SET 2	SCREEN CODE	CBM ASCII
@	@	0	64	Q	q	17	81	"	"	34	34
A	a	1	65	R	r	18	82	#	#	35	35
B	b	2	66	S	s	19	83	$	$	36	36
C	c	3	67	T	t	20	84	%	%	37	37
D	d	4	68	U	u	21	85	&	&	38	38
E	e	5	69	V	v	22	86	'	'	39	39
F	f	6	70	W	w	23	87	(	(	40	40
G	g	7	71	X	x	24	88	)	)	41	41
H	h	8	72	Y	y	25	89	*	*	42	42
I	i	9	73	Z	z	26	90	+	+	43	43
J	j	10	74	[	[	27	91	,	,	44	44
K	k	11	75	£	£	28	92	-	-	45	45
L	l	12	76	]	]	29	93	.	.	46	46
M	m	13	77	↑	↑	30	94	/	/	47	47
N	n	14	78	←	←	31	95	0	0	48	48
O	o	15	79			32	32	1	1	49	49
P	p	16	80	!	!	33	33	2	2	50	50

APPENDICE 4

8255 A

8255A/8255A-5 PROGRAMMABLE PERIPHERAL INTERFACE

- MCS-85™ Compatible 8255A-5
- 24 Programmable I/O Pins
- Completely TTL Compatible
- Fully Compatible with Intel® Microprocessor Families
- Improved Timing Characteristics
- Direct Bit Set/Reset Capability Easing Control Application Interface
- 40-Pin Dual In-Line Package
- Reduces System Package Count
- Improved DC Driving Capability

The Intel® 8255A is a general purpose programmable I/O device designed for use with Intel® microprocessors. It has 24 I/O pins which may be individually programmed in 2 groups of 12 and used in 3 major modes of operation. In the first mode (MODE 0), each group of 12 I/O pins may be programmed in sets of 4 to be input or output. In MODE 1, the second mode, each group may be programmed to have 8 lines of input or output. Of the remaining 4 pins, 3 are used for handshaking and interrupt control signals. The third mode of operation (MODE 2) is a bidirectional bus mode which uses 8 lines for a bidirectional bus, and 5 lines, borrowing one from the other group, for handshaking.

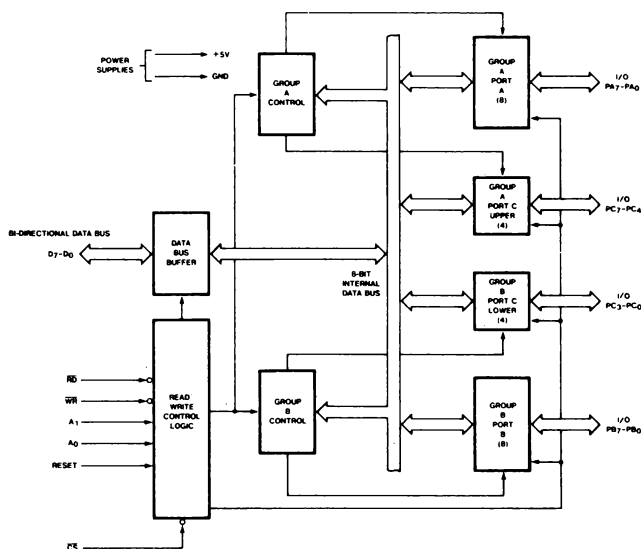



Figure 1. 8255A Block Diagram

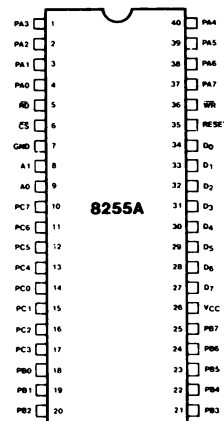



Figure 2. Pin Configuration

8255A FUNCTIONAL DESCRIPTION

General

The 8255A is a programmable peripheral interface (PPI) device designed for use in Intel® microcomputer systems. Its function is that of a general purpose I/O component to interface peripheral equipment to the microcomputer system bus. The functional configuration of the 8255A is programmed by the system software so that normally no external logic is necessary to interface peripheral devices or structures.

Data Bus Buffer

This 3-state bidirectional 8-bit buffer is used to interface the 8255A to the system data bus. Data is transmitted or received by the buffer upon execution of input or output instructions by the CPU. Control words and status information are also transferred through the data bus buffer.

Read/Write and Control Logic

The function of this block is to manage all of the internal and external transfers of both Data and Control or Status words. It accepts inputs from the CPU Address and Control busses and in turn, issues commands to both of the Control Groups.

($\overline{CS}$)

Chip Select. A "low" on this input pin enables the communiction between the 8255A and the CPU.

($\overline{RD}$)

Read. A "low" on this input pin enables the 8255A to send the data or status information to the CPU on the data bus. In essence, it allows the CPU to "read from" the 8255A.

($\overline{WR}$)

Write. A "low" on this input pin enables the CPU to write data or control words into the 8255A.

(A_0 and A_1)

Port Select 0 and Port Select 1. These input signals, in conjunction with the RD and WR inputs, control the selection of one of the three ports or the control word registers. They are normally connected to the least significant bits of the address bus (A_0 and A_1).

8255A BASIC OPERATION

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	INPUT OPERATION (READ)
0	0	0	1	0	PORT A ⇒ DATA BUS
0	1	0	1	0	PORT B ⇒ DATA BUS
1	0	0	1	0	PORT C ⇒ DATA BUS
					OUTPUT OPERATION (WRITE)
0	0	1	0	0	DATA BUS ⇒ PORT A
0	1	1	0	0	DATA BUS ⇒ PORT B
1	0	1	0	0	DATA BUS ⇒ PORT C
1	1	1	0	0	DATA BUS ⇒ CONTROL
					DISABLE FUNCTION
X	X	X	X	1	DATA BUS ⇒ 3-STATE
1	1	0	1	0	ILLEGAL CONDITION
X	X	1	1	0	DATA BUS ⇒ 3-STATE

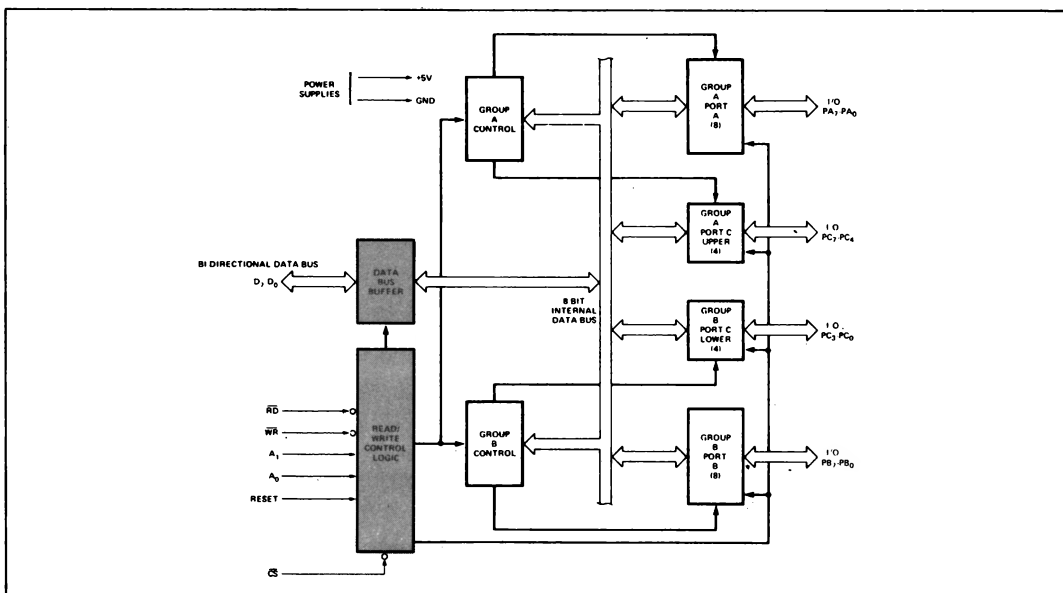



Figure 3. 8255A Block Diagram Showing Data Bus Buffer and Read/Write Control Logic Functions

(RESET)

Reset. A "high on this input clears the control register and all ports (A, C, C) are set to the input mode.

Group A and Group B Controls

The functional configuration of each port is programmed by the systems software. In essence, the CPU "outputs" a control word to the 8255A. The control word contains information such as "mode", "bit set", "bit reset", etc., that initializes the functional configuration of the 8255A.

Each of the Control blocks (Group A and Group B) accepts "commands" from the Read/Write Control Logic, receives "control words" from the internal data bus and issues the proper commands to its associated ports.

Control Group A – Port A and Port C upper (C7-C4)
Control Group B – Port B and Port C lower (C3-C0)

The Control Word Register can Only be written into. No Read operation of the Control Word Register is allowed.

Ports A, B, and C

The 8255A contains three 8-bit ports (A, B, and C). All can be configured in a wide variety of functional characteristics by the system software but each has its own special features or "personality" to further enhance the power and flexibility of the 8255A.

Port A. One 8-bit data output latch/buffer and one 8-bit data input latch.

Port B. One 8-bit data input/output latch/buffer and one 8-bit data input buffer.

Port C. One 8-bit data output latch/buffer and one 8-bit data input buffer (no latch for input). This port can be divided into two 4-bit ports under the mode control. Each 4-bit port contains a 4-bit latch and it can be used for the control signal outputs and status signal inputs in conjunction with ports A and B.

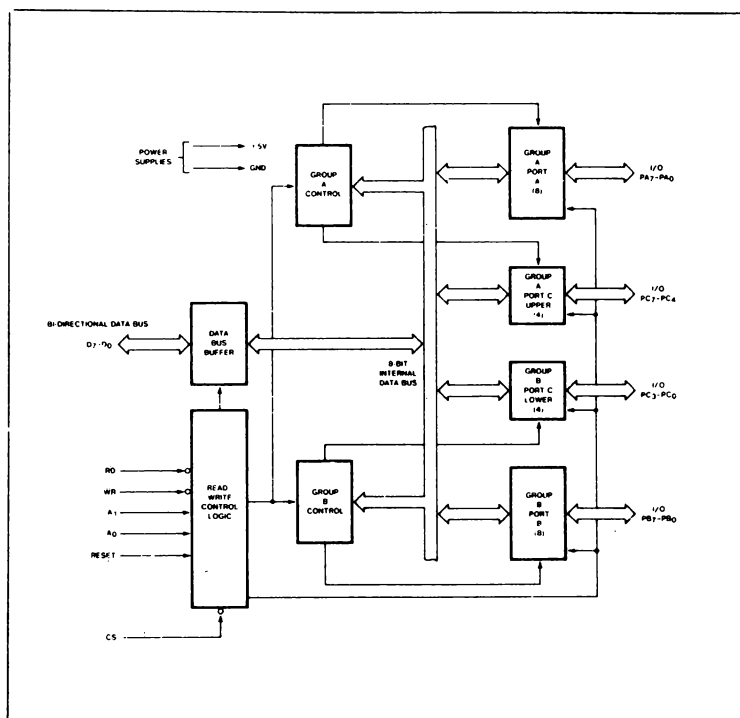
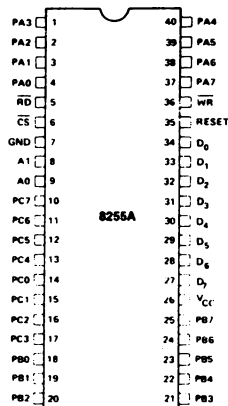



Figure 4. 8225A Block Diagram Showing Group A and Group B Control Functions

PIN CONFIGURATION

8255A

Pin	Name	Pin	Name
1	PA3	40	PA4
2	PA2	39	PA5
3	PA1	38	PA6
4	PA0	37	PA7
5	RD	36	WR
6	CS	35	RESET
7	GND	34	D_0
8	A1	33	D_1
9	A0	32	D_2
10	PC7	31	D_3
11	PC6	30	D_4
12	PC5	29	D_5
13	PC4	28	D_6
14	PC0	27	D_7
15	PC1	26	V_{CC}
16	PC2	25	PB7
17	PC3	24	PB6
18	PB0	23	PB5
19	PB1	22	PB4
20	PB2	21	PB3

PIN NAMES

D_7-D_0	DATA BUS (BI-DIRECTIONAL)
RESET	RESET INPUT
CS	CHIP SELECT
RD	READ INPUT
WR	WRITE INPUT
A0, A1	PORT ADDRESS
PA7-PA0	PORT A (BIT)
PB7-PB0	PORT B (BIT)
PC7-PC0	PORT C (BIT)
V_{CC}	+5 VOLTS
GND	0 VOLTS

8255A OPERATIONAL DESCRIPTION

Mode Selection

There are three basic modes of operation that can be selected by the system software:

Mode 0 — Basic Input/Output
Mode 1 — Strobed Input/Output
Mode 2 — Bi-Directional Bus

When the reset input goes "high" all ports will be set to the input mode (i.e., all 24 lines will be in the high impedance state). After the reset is removed the 8255A can remain in the input mode with no additional initialization required. During the execution of the system program any of the other modes may be selected using a single output instruction. This allows a single 8255A to service a variety of peripheral devices with a simple software maintenance routine.

The modes for Port A and Port B can be separately defined, while Port C is divided into two portions as required by the Port A and Port B definitions. All of the output registers, including the status flip-flops, will be reset whenever the mode is changed. Modes may be combined so that their functional definition can be "tailored" to almost any I/O structure. For instance; Group B can be programmed in Mode 0 to monitor simple switch closings or display computational results, Group A could be programmed in Mode 1 to monitor a keyboard or tape reader on an interrupt-driven basis.

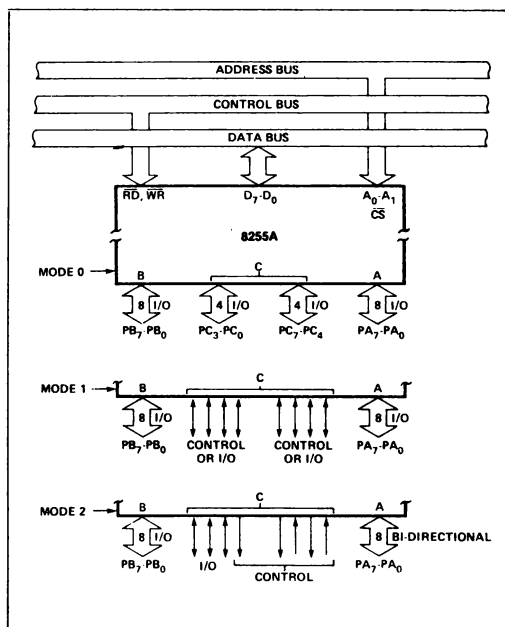



Figure 5. Basic Mode Definitions and Bus Interface

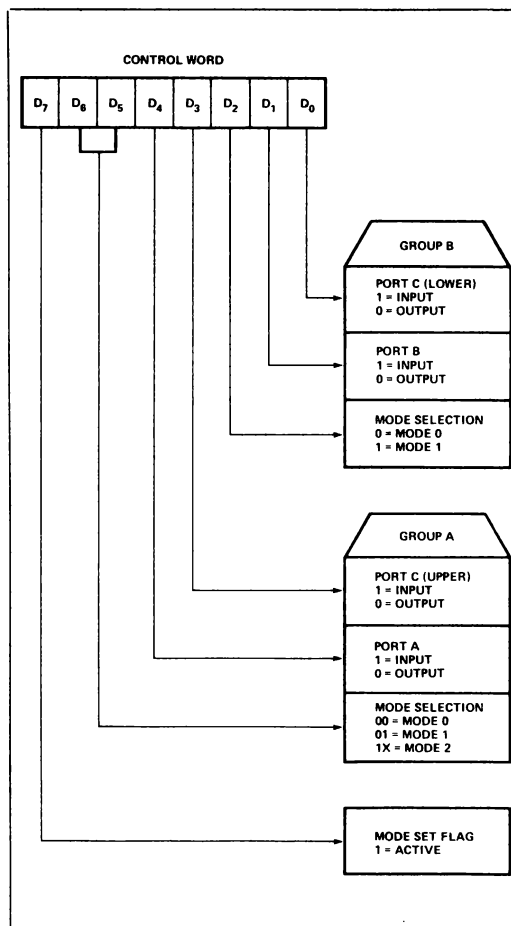



Figure 6. Mode Definition Format

The mode definitions and possible mode combinations may seem confusing at first but after a cursory review of the complete device operation a simple, logical I/O approach will surface. The design of the 8255A has taken into account things such as efficient PC board layout, control signal definition vs PC layout and complete functional flexibility to support almost any peripheral device with no external logic. Such design represents the maximum use of the available pins.

Single Bit Set/Reset Feature

Any of the eight bits of Port C can be Set or Reset using a single OUTput instruction. This feature reduces software requirements in Control-based applications.

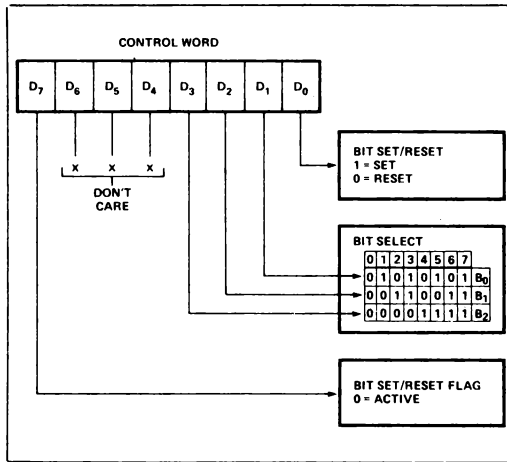



Figure 7. Bit Set/Reset Format

When Port C is being used as status/control for Port A or B, these bits can be set or reset by using the Bit Set/Reset operation just as if they were data output ports.

Interrupt Control Functions

When the 8255A is programmed to operate in mode 1 or mode 2, control signals are provided that can be used as interrupt request inputs to the CPU. The interrupt request signals, generated from port C, can be inhibited or enabled by setting or resetting the associated INTE flip-flop, using the bit set/reset function of port C.

This function allows the Programmer to disallow or allow a specific I/O device to interrupt the CPU without affecting any other device in the interrupt structure.

INTE flip-flop definition:

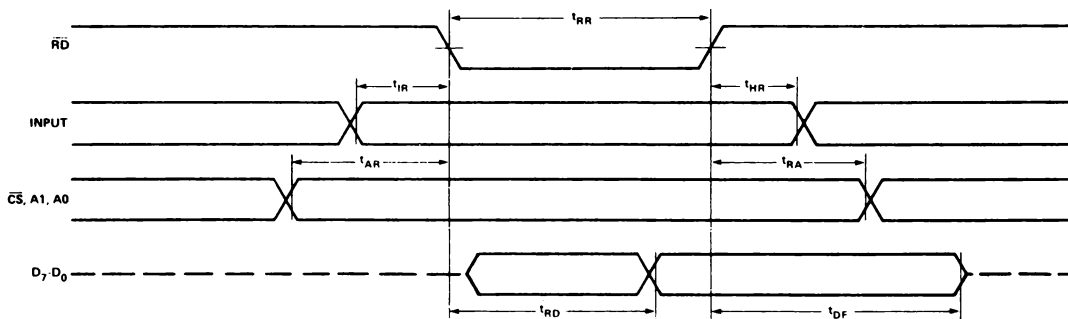
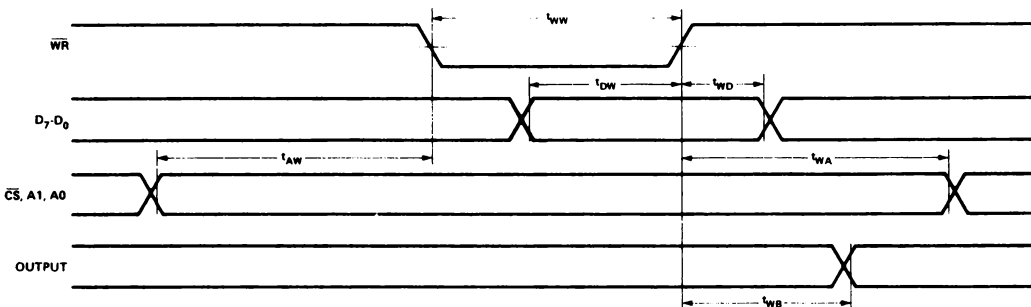
(BIT-SET) – INTE is SET – Interrupt enable
(BIT-RESET) – INTE is RESET – Interrupt disable

Note: All Mask flip-flops are automatically reset during mode selection and device Reset.

Operating Modes

MODE 0 (Basic Input/Output). This functional configuration provides simple input and output operations for each of the three ports. No "handshaking" is required, data is simply written to or read from a specified port.

Mode 0 Basic Functional Definitions:

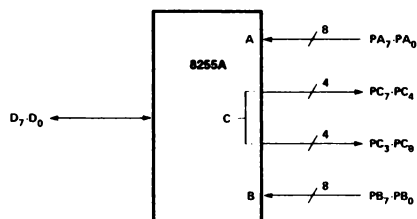
- Two 8-bit ports and two 4-bit ports.
- Any port can be input or output.
- Outputs are latched.
- Inputs are not latched.
- 16 different Input/Output configurations are possible in this Mode.

MODE 0 (Basic Input)

MODE 0 (Basic Output)

8255A/8255A-5

MODE 0 Port Definition

A		B		GROUP A			GROUP B	
D_4	D_3	D_1	D_0	PORT A	PORT C (UPPER)	#	PORT B	PORT C (LOWER)
0	0	0	0	OUTPUT	OUTPUT	0	OUTPUT	OUTPUT
0	0	0	1	OUTPUT	OUTPUT	1	OUTPUT	INPUT
0	0	1	0	OUTPUT	OUTPUT	2	INPUT	OUTPUT
0	0	1	1	OUTPUT	OUTPUT	3	INPUT	INPUT
0	1	0	0	OUTPUT	INPUT	4	OUTPUT	OUTPUT
0	1	0	1	OUTPUT	INPUT	5	OUTPUT	INPUT
0	1	1	0	OUTPUT	INPUT	6	INPUT	OUTPUT
0	1	1	1	OUTPUT	INPUT	7	INPUT	INPUT
1	0	0	0	INPUT	OUTPUT	8	OUTPUT	OUTPUT
1	0	0	1	INPUT	OUTPUT	9	OUTPUT	INPUT
1	0	1	0	INPUT	OUTPUT	10	INPUT	OUTPUT
1	0	1	1	INPUT	OUTPUT	11	INPUT	INPUT
1	1	0	0	INPUT	INPUT	12	OUTPUT	OUTPUT
1	1	0	1	INPUT	INPUT	13	OUTPUT	INPUT
1	1	1	0	INPUT	INPUT	14	INPUT	OUTPUT
1	1	1	1	INPUT	INPUT	15	INPUT	INPUT

MODE 0 Configurations

CONTROL WORD #4
D7 D6 D5 D4 D3 D2 D1 D0
1 0 0 0 1 0 0 0
8255A
A
C
B
D7-D0
PA7-PA0
PC7-PC4
PC3-PC0
PB7-PB0
CONTROL WORD #8
1 0 0 1 0 0 0 0
CONTROL WORD #5
1 0 0 0 1 0 0 1
CONTROL WORD #9
1 0 0 1 0 0 0 1
CONTROL WORD #6
1 0 0 0 1 0 1 0
CONTROL WORD #10
1 0 0 1 0 0 1 0
CONTROL WORD #7
1 0 0 0 1 0 1 1
CONTROL WORD #11
1 0 0 1 0 0 1 1

8255A/8255A-5

CONTROL WORD #12

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	1	1	0	0	0

8255A — D_7-D_0; A: 8, PA_7-PA_0; C: 4, PC_7-PC_4; 4, PC_3-PC_0; B: 8, PB_7-PB_0

CONTROL WORD #13

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	1	1	0	0	1

8255A — D_7-D_0; A: 8, PA_7-PA_0; C: 4, PC_7-PC_4; 4, PC_3-PC_0; B: 8, PB_7-PB_0

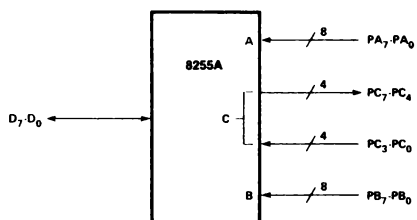
CONTROL WORD #14

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	1	1	0	1	0

8255A — D_7-D_0; A: 8, PA_7-PA_0; C: 4, PC_7-PC_4; 4, PC_3-PC_0; B: 8, PB_7-PB_0

CONTROL WORD #15

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	1	1	0	1	1

8255A — D_7-D_0; A: 8, PA_7-PA_0; C: 4, PC_7-PC_4; 4, PC_3-PC_0; B: 8, PB_7-PB_0

Operating Modes

MODE 1 (Strobed Input/Output). This functional configuration provides a means for transferring I/O data to or from a specified port in conjunction with strobes or "handshaking" signals. In mode 1, port A and Port B use the lines on port C to generate or accept these "handshaking" signals.

Mode 1 Basic Functional Definitions:

- Two Groups (Group A and Group B)
- Each group contains one 8-bit data port and one 4-bit control/data port.
- The 8-bit data port can be either input or output. Both inputs and outputs are latched.
- The 4-bit port is used for control and status of the 8-bit data port.

8255A/8255A-5

Input Control Signal Definition

STB (Strobe Input). A "low" on this input loads data into the input latch.

IBF (Input Buffer Full F/F)

A "high" on this output indicates that the data has been loaded into the input latch; in essence, an acknowledgement. IBF is set by STB input being low and is reset by the rising edge of the RD input.

INTR (Interrupt Request)

A "high" on this output can be used to interrupt the CPU when an input device is requesting service. INTR is set by the $\overline{STB}$ is a "one", IBF is a "one" and INTE is a "one". It is reset by the falling edge of $\overline{RD}$. This procedure allows an input device to request service from the CPU by simply strobing its data into the port.

INTE A

Controlled by bit set/reset of PC_4.

INTE B

Controlled by bit set/reset of PC_2.

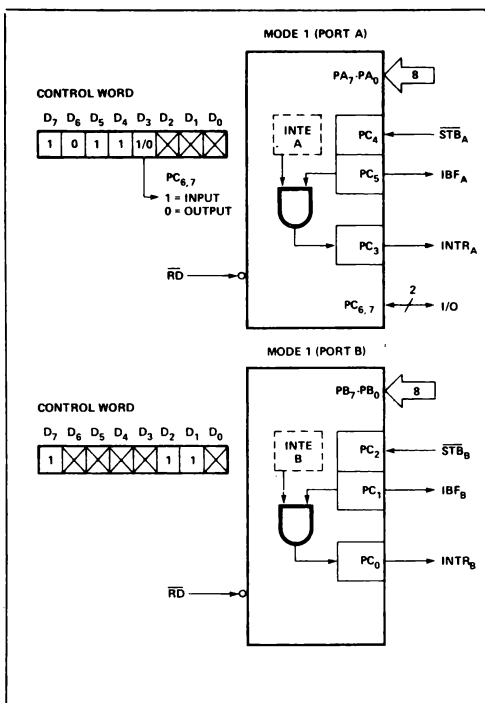



Figure 8. MODE 1 Input

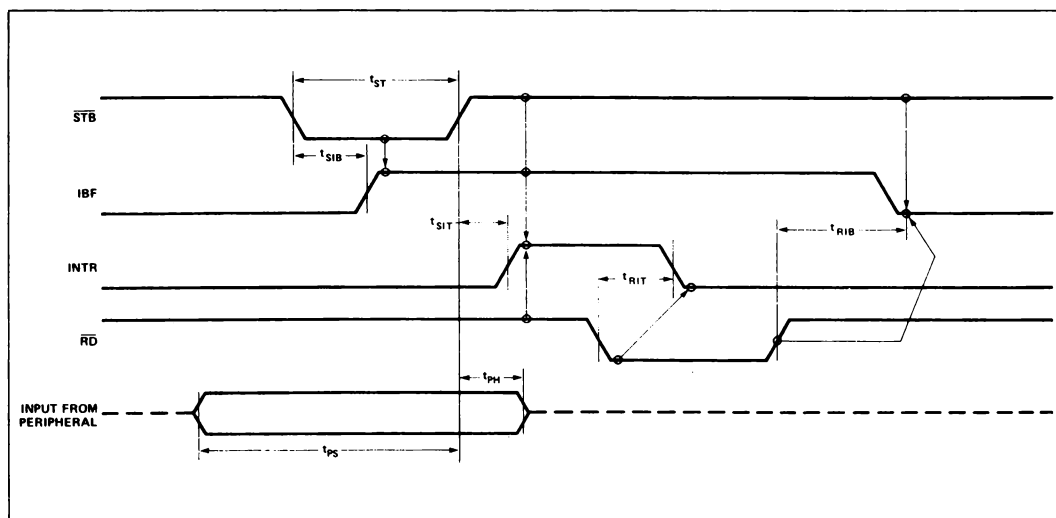



Figure 9. MODE 1 (Strobed Input)

APPENDICE 5

ADC 0809

Analog-to-Digital Converters

ADC0808, ADC0809 8-Bit μP Compatible A/D Converters With 8-Channel Multiplexer

General Description

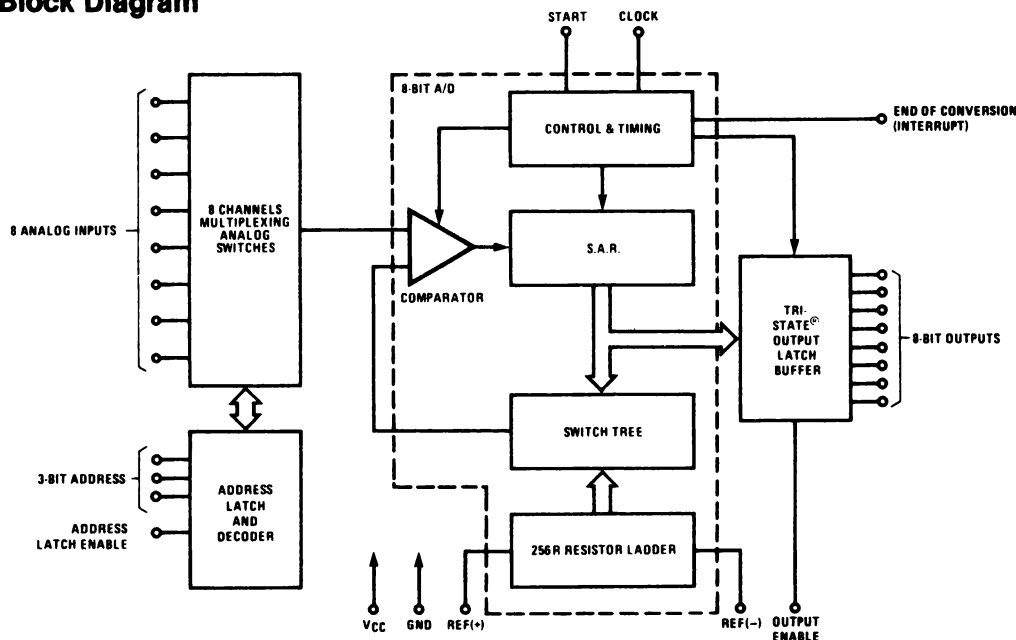
The ADC0808, ADC0809 data acquisition component is a monolithic CMOS device with an 8-bit analog-to-digital converter, 8-channel multiplexer and microprocessor compatible control logic. The 8-bit A/D converter uses successive approximation as the conversion technique. The converter features a high impedance chopper stabilized comparator, a 256R voltage divider with analog switch tree and a successive approximation register. The 8-channel multiplexer can directly access any of 8-single-ended analog signals.

The device eliminates the need for external zero and full-scale adjustments. Easy interfacing to microprocessors is provided by the latched and decoded multiplexer address inputs and latched TTL TRI-STATE® outputs.

The design of the ADC0808, ADC0809 has been optimized by incorporating the most desirable aspects of several A/D conversion techniques. The ADC0808, ADC0809 offers high speed, high accuracy, minimal temperature dependence, excellent long-term accuracy and repeatability, and consumes minimal power. These features make this device ideally suited to applications from process and machine control to consumer and automotive applications. For 16-channel multiplexer with common output (sample/hold port) see ADC0816 data sheet.

Features

- Resolution — 8-bits
- Total unadjusted error — ± 1/2 LSB and ± 1 LSB
- No missing codes
- Conversion time — 100 μs
- Single supply — 5 V_{DC}
- Operates ratiometrically or with 5 V_{DC} or analog span adjusted voltage reference
- 8-channel multiplexer with latched control logic
- Easy interface to all microprocessors, or operates "stand alone"
- Outputs meet T^2L voltage level specifications
- 0V to 5V analog input voltage range with single 5V supply
- No zero or full-scale adjust required
- Standard hermetic or molded 28-pin DIP package
- Temperature range −40°C to +85°C or −55°C to +125°C
- Low power consumption — 15 mW
- Latched TRI-STATE® output

Block Diagram

Absolute Maximum Ratings (Notes 1 and 2)

Supply Voltage (V_{CC}) (Note 3)	6.5V
Voltage at Any Pin Except Control Inputs	$-0.3V$ to ($V_{CC}+0.3V$)
Voltage at Control Inputs (START, OE, CLOCK, ALE, ADD A, ADD B, ADD C)	$-0.3V$ to $+15V$
Storage Temperature Range	$-65°C$ to $+150°C$
Package Dissipation at $T_A = 25°C$	875 mW
Lead Temperature (Soldering, 10 seconds)	300°C

Operating Ratings (Notes 1 and 2)

Temperature Range (Note 1)	$T_{MIN} \le T_A \le T_{MAX}$
ADC0808CJ	$-55°C \le T_A \le +125°C$
ADC0808CCJ, ADC0808CCN, ADC0809CCN	$-40°C \le T_A \le +85°C$
Range of V_{CC} (Note 1)	$4.5\,V_{DC}$ to $6.0\,V_{DC}$

Electrical Characteristics

Converter Specifications: $V_{CC} = 5\,V_{DC} = V_{REF(+)}$, $V_{REF(-)} = GND$, $T_{MIN} \le T_A \le T_{MAX}$ and $f_{CLK} = 640$ kHz unless otherwise stated.

	Parameter	Conditions	Min	Typ	Max	Units
	ADC0808					
	Total Unadjusted Error	25°C			± 1/2	LSB
	(Note 5)	T_{MIN} to T_{MAX}			± 3/4	LSB
	ADC0809					
	Total Unadjusted Error	0°C to 70°C			± 1	LSB
	(Note 5)	T_{MIN} to T_{MAX}			± 1 1/4	LSB
	Input Resistance	From Ref(+) to Ref(−)	1.0	2.5		kΩ
	Analog Input Voltage Range	(Note 4) V(+) or V(−)	GND-0.10		V_{CC}+0.10	V_{DC}
$V_{REF(+)}$	Voltage, Top of Ladder	Measured at Ref(+)		V_{CC}	V_{CC}+0.1	V
$\frac{V_{REF(+)}+V_{REF(-)}}{2}$	Voltage, Center of Ladder		V_{CC}/2-0.1	V_{CC}/2	V_{CC}/2+0.1	V
$V_{REF(-)}$	Voltage, Bottom of Ladder	Measured at Ref(−)	−0.1	0		V
	Comparator Input Current	$f_c = 640$ kHz, (Note 6)	−2	±0.5	2	μA

Electrical Characteristics

Digital Levels and DC Specifications: ADC0808CJ $4.5V \le V_{CC} \le 5.5V$, $-55°C \le T_A \le +125°C$ unless otherwise noted ADC0808CCJ, ADC0808CCN, and ADC0809CCN $4.75 \le V_{CC} \le 5.25V$, $-40°C \le T_A \le +85°C$ unless otherwise noted

	Parameter	Conditions	Min	Typ	Max	Units
ANALOG MULTIPLEXER						
$I_{OFF(+)}$	OFF Channel Leakage Current	$V_{CC} = 5V$, $V_{IN} = 5V$,				
		$T_A = 25°C$		10	200	nA
		T_{MIN} to T_{MAX}			1.0	μA
$I_{OFF(-)}$	OFF Channel Leakage Current	$V_{CC} = 5V$, $V_{IN} = 0$,				
		$T_A = 25°C$	−200	−10		nA
		T_{MIN} to T_{MAX}	−1.0			μA
CONTROL INPUTS						
$V_{IN(1)}$	Logical "1" Input Voltage		V_{CC}-1.5			V
$V_{IN(0)}$	Logical "0" Input Voltage				1.5	V
$I_{IN(1)}$	Logical "1" Input Current (The Control Inputs)	$V_{IN} = 15V$			1.0	μA
$I_{IN(0)}$	Logical "0" Input Current (The Control Inputs)	$V_{IN} = 0$	−1.0			μA
I_{CC}	Supply Current	$f_{CLK} = 640$ kHz		0.3	3.0	mA

Electrical Characteristics (Continued)

Digital Levels and DC Specifications: ADC0808CJ $4.5V \leq V_{CC} \leq 5.5V$, $-55°C \leq T_A \leq +125°C$ unless otherwise noted ADC0808CCJ, ADC0808CCN, and ADC0809CCN $4.75 \leq V_{CC} \leq 5.25V$, $-40°C \leq T_A \leq +85°C$ unless otherwise noted

Parameter		Conditions	Min	Typ	Max	Units
DATA OUTPUTS AND EOC (INTERRUPT)						
$V_{OUT(1)}$	Logical "1" Output Voltage	$I_O = -360\ \mu A$	V_{CC}-0.4			V
$V_{OUT(0)}$	Logical "0" Output Voltage	$I_O = 1.6$ mA			0.45	V
$V_{OUT(0)}$	Logical "0" Output Voltage EOC	$I_O = 1.2$ mA			0.45	V
I_{OUT}	TRI-STATE Output Current	$V_O = 5V$			3	μA
		$V_O = 0$	−3			μA

Electrical Characteristics

Timing Specifications: $V_{CC} = V_{REF(+)} = 5V$, $V_{REF(-)} = GND$, $t_r = t_f = 20$ ns and $T_A = 25°C$ unless otherwise noted.

Symbol	Parameter	Conditions	Min	Typ	Max	Units
t_{WS}	Minimum Start Pulse Width	*(Figure 5)*		100	200	ns
t_{WALE}	Minimum ALE Pulse Width	*(Figure 5)*		100	200	ns
t_s	Minimum Address Set-Up Time	*(Figure 5)*		25	50	ns
t_H	Minimum Address Hold Time	*(Figure 5)*		25	50	ns
t_D	Analog MUX Delay Time From ALE	$R_S = 0\Omega$ *(Figure 5)*		1	2.5	μs
t_{H1}, t_{H0}	OE Control to Q Logic State	$C_L = 50$ pF, $R_L = 10k$ *(Figure 8)*		125	250	ns
t_{1H}, t_{0H}	OE Control to Hi-Z	$C_L = 10$ pF, $R_L = 10k$ *(Figure 8)*		125	250	ns
t_c	Conversion Time	$f_c = 640$ kHz, *(Figure 5)* (Note 7)	90	100	116	μs
f_c	Clock Frequency		10	640	1280	kHz
t_{EOC}	EOC Delay Time	*(Figure 5)*	0		8+2 μs	Clock Periods
C_{IN}	Input Capacitance	At Control Inputs		10	15	pF
C_{OUT}	TRI-STATE® Output Capacitance	At TRI-STATE® Outputs, (Note 12)		10	15	pF

Note 1: Absolute maximum ratings are those values beyond which the life of the device may be impaired.

Note 2: All voltages are measured with respect to GND, unless otherwise specified.

Note 3: A zener diode exists, internally, from V_{CC} to GND and has a typical breakdown voltage of 7 V_{DC}.

Note 4: Two on-chip diodes are tied to each analog input which will forward conduct for analog input voltages one diode drop below ground or one diode drop greater than the V_{CC} supply. The spec allows 100 mV forward bias of either diode. This means that as long as the analog V_{IN} does not exceed the supply voltage by more than 100 mV, the output code will be correct. To achieve an absolute 0 V_{DC} to 5 V_{DC} input voltage range will therefore require a minimum supply voltage of 4.900 V_{DC} over temperature variations, initial tolerance and loading.

Note 5: Total unadjusted error includes offset, full-scale, linearity, and multiplexer errors. See *Figure 3*. None of these A/Ds requires a zero or full-scale adjust. However, if an all zero code is desired for an analog input other than 0.0V, or if a narrow full-scale span exists (for example: 0.5V to 4.5V full-scale) the reference voltages can be adjusted to achieve this. See *Figure 13*.

Note 6: Comparator input current is a bias current into or out of the chopper stabilized comparator. The bias current varies directly with clock frequency and has little temperature dependence *(Figure 6)*. See paragraph 4.0.

Note 7: The outputs of the data register are updated one clock cycle before the rising edge of EOC.

APPENDICE 6

LM 35

National Semiconductor

Sensors/Transducers

LM135/LM235/LM335, LM135A/LM235A/LM335A Precision Temperature Sensors

General Description

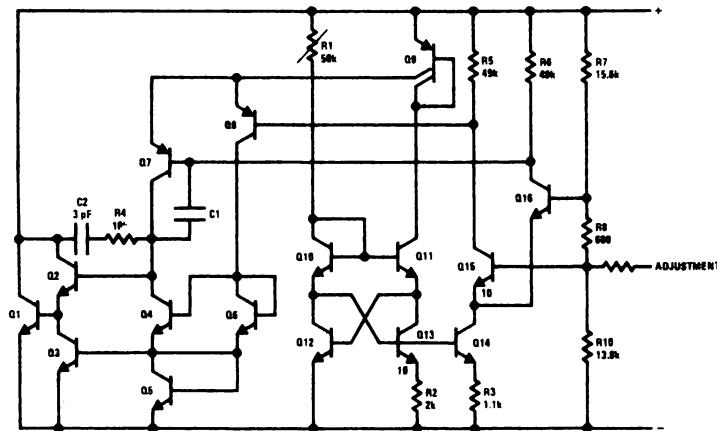
The LM135 series are precision, easily-calibrated, integrated circuit temperature sensors. Operating as a 2-terminal zener, the LM135 has a breakdown voltage directly proportional to absolute temperature at +10 mV/°K. With less than 1Ω dynamic impedance the device operates over a current range of 400 μA to 5 mA with virtually no change in performance. When calibrated at 25°C the LM135 has typically less than 1°C error over a 100°C temperature range. Unlike other sensors the LM135 has a linear output.

Applications for the LM135 include almost any type of temperature sensing over a −55°C to +150°C temperature range. The low impedance and linear output make interfacing to readout or control circuitry especially easy.

The LM135 operates over a −55°C to +150°C temperature range while the LM235 operates over a −40°C to +125°C temperature range. The LM335 operates from −10°C to +100°C. The LM135/LM235/LM335 are available packaged in hermetic TO-46 transistor packages while the LM235 and LM335 are also available in plastic TO-92 packages.

Features

- Directly calibrated in °Kelvin
- 1°C initial accuracy available
- Operates from 400 μA to 5 mA
- Less than 1Ω dynamic impedance
- Easily calibrated
- Wide operating temperature range
- 200°C overrange
- Low cost

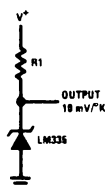
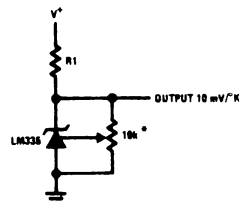
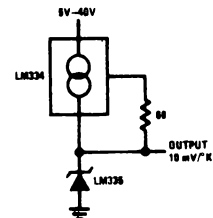
Schematic Diagram

Typical Applications

Basic Temperature Sensor

V+
R1
OUTPUT 10 mV/°K
LM335

Calibrated Sensor

V+
R1
OUTPUT 10 mV/°K
LM335
10k*

*Calibrate for 2.982V at 25°C

Wide Operating Supply

Absolute Maximum Ratings

Reverse Current		10 mA
Forward Current		10 mA
Storage Temperature		
TO-46 Package		−60°C to +180°C
TO-92 Package		−60°C to +150°C
Specified Operating Temperature Range		
	Continuous	**Intermittent**
LM135, LM135A	−55°C to +150°C	150°C to 200°C
LM235, LM235A	−40°C to +125°C	125°C to 150°C
LM335, LM335A	−10°C to +100°C	100°C to 125°C
Lead Temperature (Soldering, 10 seconds)		300°C

Temperature Accuracy LM135/LM235, LM135A/LM235A (Note 1)

PARAMETER	CONDITIONS	LM135A/LM235A			LM135/LM235			UNITS
		MIN	TYP	MAX	MIN	TYP	MAX	
Operating Output Voltage	$T_C = 25°C, I_R = 1$ mA	2.97	2.98	2.99	2.95	2.98	3.01	V
Uncalibrated Temperature Error	$T_C = 25°C, I_R = 1$ mA		0.5	1		1	3	°C
Uncalibrated Temperature Error	$T_{MIN} < T_C < T_{MAX}, I_R = 1$ mA		1.3	2.7		2	5	°C
Temperature Error with 25°C Calibration	$T_{MIN} < T_C < T_{MAX}, I_R = 1$ mA		0.3	1		0.5	1.5	°C
Calibrated Error at Extended Temperatures	$T_C = T_{MAX}$ (Intermittent)		2			2		°C
Non-Linearity	$I_R = 1$ mA		0.3	0.5		0.3	1	°C

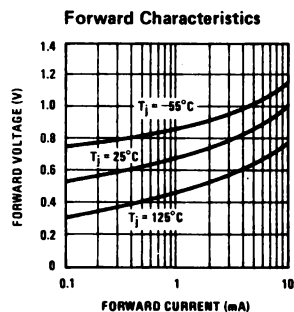
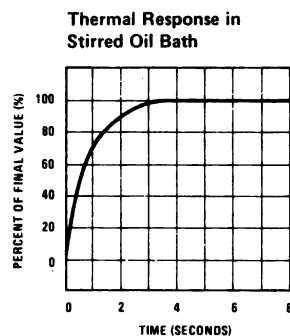
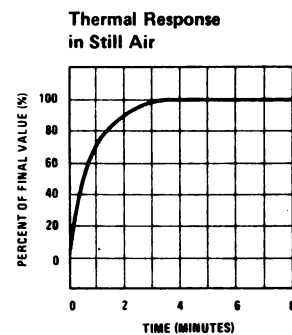
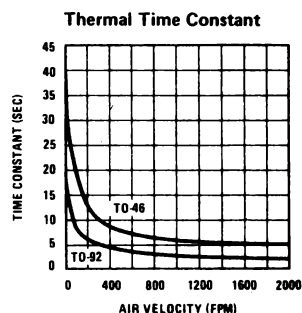
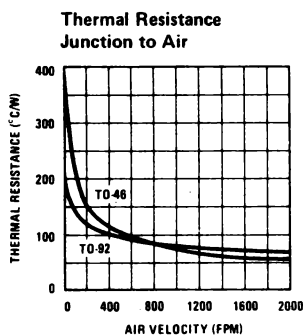
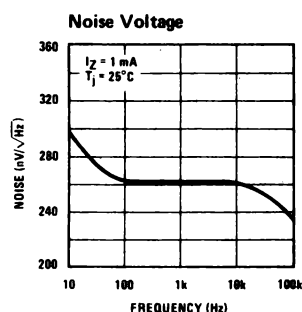
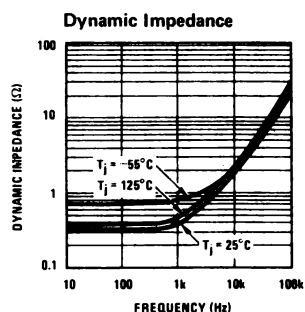
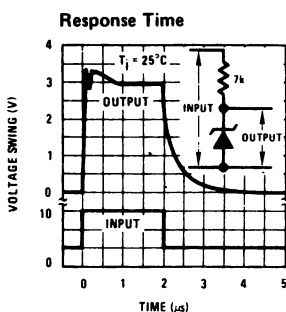
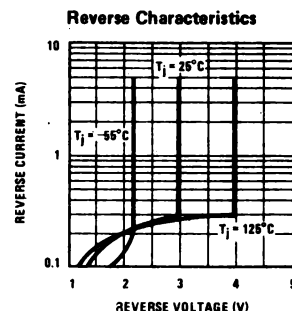
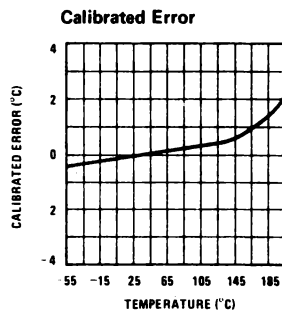
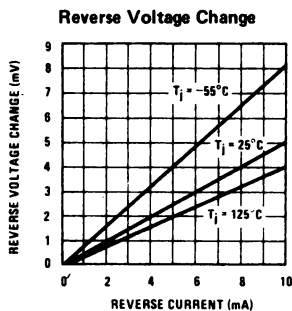
Temperature Accuracy LM335, LM335A (Note 1)

PARAMETER	CONDITIONS	LM335A			LM335			UNITS
		MIN	TYP	MAX	MIN	TYP	MAX	
Operating Output Voltage	$T_C = 25°C, I_R = 1$ mA	2.95	2.98	3.01	2.92	2.98	3.04	V
Uncalibrated Temperature Error	$T_C = 25°C, I_R = 1$ mA		1	3		2	6	°C
Uncalibrated Temperature Error	$T_{MIN} < T_C < T_{MAX}, I_R = 1$ mA		2	5		4	9	°C
Temperature Error with 25°C Calibration	$T_{MIN} < T_C < T_{MAX}, I_R = 1$ mA		0.5	1		1	2	°C
Calibrated Error at Extended Temperatures	$T_C = T_{MAX}$ (Intermittent)		2			2		°C
Non-Linearity	$I_R = 1$ mA		0.3	1.5		0.3	1.5	°C

Electrical Characteristics (Note 1)

PARAMETER	CONDITIONS	LM135/LM235 LM135A/LM235A			LM335 LM335A			UNITS
		MIN	TYP	MAX	MIN	TYP	MAX	
Operating Output Voltage Change with Current	400 μA $< I_R <$ 5 mA At Constant Temperature		2.5	10		3	14	mV
Dynamic Impedance	$I_R = 1$ mA		0.5			0.6		Ω
Output Voltage Temperature Drift			+10			+10		mV/°C
Time Constant	Still Air		80			80		sec
	100 ft/Min Air		10			10		sec
	Stirred Oil		1			1		sec
Time Stability	$T_C = 125°C$		0.2			0.2		°C/khr

Note 1: Accuracy measurements are made in a well-stirred oil bath. For other conditions, self heating must be considered.

Typical Performance Characteristics

Definition of Terms

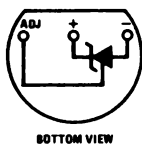
Operating Output Voltage: The voltage appearing across the positive and negative terminals of the device at specified conditions of operating temperature and current.

Uncalibrated Temperature Error: The error between the operating output voltage at 10 mV/°K and case temperature at specified conditions of current and case temperature.

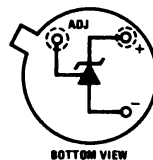
Calibrated Temperature Error: The error between operating output voltage and case temperature at 10 mV/°K over a temperature range at a specified operating current with the 25°C error adjusted to zero.

Connection Diagrams

TO-92
Plastic Package

BOTTOM VIEW

Order Number LM235Z, LM335Z or LM335AZ
See NS Package Z03A

TO-46
Metal Can Package*

BOTTOM VIEW

*Case is connected to negative pin

Order Number LM135H, LM235H, LM335H, LM135AH, LM235AH or LM335AH
See NS Package H03H

APPENDICE 7

2732 A

2732A
32K (4K x 8) UV ERASABLE PROM

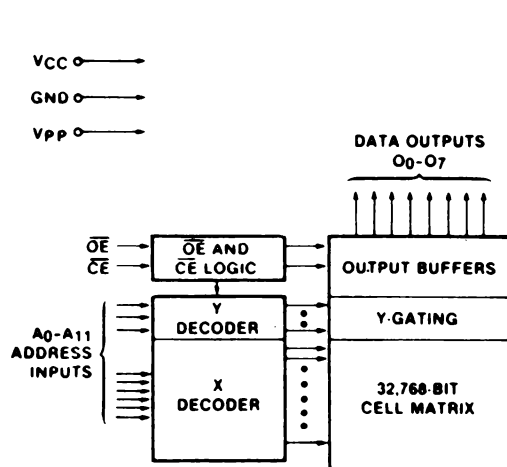
- **200 ns (2732A-2) Maximum Access Time . . . HMOS*-E Technology**
- **Compatible to High Speed 8 MHz 8086-2 MPU . . . Zero WAIT State**
- **Two Line Control**
- **Pin Compatible to 2764 EPROM**
- **Industry Standard Pinout . . . JEDEC Approved**
- **Low Standby Current . . . 35 mA Maximum**
- **±10% V_{CC} Tolerance Available**

The Intel® 2732A is a 5V only, 32,768 bit ultraviolet erasable and electrically programmable read-only memory (EPROM). It is pin compatible to Intel's 450 ns 2732. The standard 2732A's access time is 250 ns with speed selection (2732A-2) available at 200 ns. The access time is compatible to high performance microprocessors, such as the 8 MHz 8086-2. In these systems, the 2732A allows the microprocessor to operate without the addition of WAIT states.

An important 2732A feature is the separate output control, Output Enable ($\overline{OE}$), from the Chip Enable control ($\overline{CE}$). The $\overline{OE}$ control eliminates bus contention in multiple bus microprocessor systems. Intel's Application Note AP-72 describes the microprocessor system implementation of the $\overline{OE}$ and $\overline{CE}$ controls on Intel's EPROMs. AP-72 is available from Intel's Literature Department.

The 2732A has a standby mode which reduces the power dissipation without increasing access time. The maximum active current is 125 mA, while the maximum standby current is only 35 mA, a 70% saving. The standby mode is achieved by applying a TTL-high signal to the $\overline{CE}$ input.

The 2732A is fabricated with HMOS*-E technology, Intel's high-speed N-channel MOS Silicon Gate Technology.

*HMOS is a patented process of Intel Corporation.

PIN NAMES

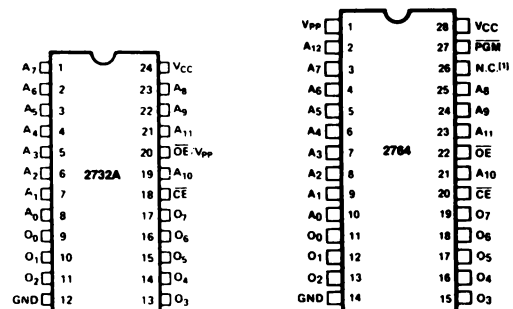
A_0-A_{11}	ADDRESSES
$\overline{CE}$	CHIP ENABLE
$\overline{OE}$	OUTPUT ENABLE
O_0-O_7	OUTPUTS

[1] For upgradability to JEDEC approved 128K EPROMs, provide an address line to pin 26. For compatibility with the 2732A and 32K ROMs, provide a trace from V_{CC} to pin 26

Figure 1. Block Diagram

Figure 2. Pin Configurations

ERASURE CHARACTERISTICS

The erasure characteristics of the 2732A are such that erasure begins to occur when exposed to light with wavelengths shorter than approximately 4000 Angstroms (Å). It should be noted that sunlight and certain types of fluorescent lamps have wavelengths in the 3000-4000Å range. Data show that constant exposure to room level fluorescent lighting could erase the typical 2732A in approximately 3 years, while it would take approximately 1 week to cause erasure when exposed to direct sunlight. If the 2732A is to be exposed to these types of lighting conditions for extended periods of time, opaque labels are available from Intel which should be placed over the 2732A window to prevent unintentional erasure.

The recommended erasure procedure for the 2732A is exposure to shortwave ultraviolet light which has a wavelength of 2537 Angstroms (Å). The integrated dose (i.e., UV intensity X exposure time) for erasure should be a minimum of 15 W-sec/cm^2. The erasure time with this dosage is approximately 15 to 20 minutes using an ultraviolet lamp with 12000μW/cm^2 power rating. The 2732A should be placed within 1 inch of the lamp tubes during erasure. Some lamps have a filter on their tubes which should be removed before erasure.

DEVICE OPERATION

The five modes of operation of the 2732A are listed in Table 1. A single 5V power supply is required in the read mode. All inputs are TTL levels except for $\overline{OE}/V_{PP}$ during programming. In the program mode the $\overline{OE}/V_{PP}$ input is pulsed from a TTL level to 21V.

Table 1. Mode Selection

PINS / MODE	$\overline{CE}$ (18)	$\overline{OE}/V_{PP}$ (20)	V_{CC} (24)	OUTPUTS (9-11,13-17)
Read	V_{IL}	V_{IL}	+5	D_{OUT}
Standby	V_{IH}	Don't Care	+5	High Z
Program	V_{IL}	V_{PP}	+5	D_{IN}
Program Verify	V_{IL}	V_{IL}	+5	D_{OUT}
Program Inhibit	V_{IH}	V_{PP}	+5	High Z

Read Mode

The 2732A has two control functions, both of which must be logically active in order to obtain data at the outputs. Chip Enable ($\overline{CE}$) is the power control and should be used for device selection. Output Enable ($\overline{OE}$) is the output control and should be used to gate data to the output pins, independent of device selection. Assuming that addresses are stable, address access time (t_{ACC}) is equal to the delay from $\overline{CE}$ to output (t_{CE}). Data is available at the outputs after the falling edge of $\overline{OE}$, assuming that $\overline{CE}$ has been low and addresses have been stable for at least t_{ACC}—t_{OE}.

Standby Mode

The 2732A has a standby mode which reduces the active power current from 125 mA to 35 mA. The 2732A is placed in the standby mode by applying a TTL-high signal to the $\overline{CE}$ input. When in standby mode, the outputs are in a high impedance state, independent of the $\overline{OE}$ input.

Output OR-Tieing

Because EPROMs are usually used in larger memory arrays, Intel has provided a 2 line control function that accommodates this use of multiple memory connection. The two line control function allows for:

a) the lowest possible memory power dissipation, and
b) complete assurance that output bus contention will not occur.

To use these two control lines most efficiently, it is recommended that $\overline{CE}$ (pin 18) be decoded and used as the primary device selecting function, while $\overline{OE}$ (pin 20) be made a common connection to all devices in the array and connected to the READ line from the system control bus. This assures that all deselected memory devices are in their low power standby mode and that the output pins are only active when data is desired from a particular memory device.

PROGRAMMING (See Programming Instruction Section for Waveforms.)

Programming is the same as Intel's 450 ns 2732 except for the programming voltage. In the program mode the 2732A $\overline{OE}/V_{PP}$ input is pulsed from a TTL low level to 21V (25V for the 2732). ***Exceeding 22V will damage the 2732A.***

Initially, and after each erasure, all bits of the 2732A are in the "1" state. Data is introduced by selectively programming "0's" into the desired bit locations. Although only "0's" will be programmed, both "1's" and "0's" can be present in the data word. The only way to change a "0" to a "1" is by ultraviolet light erasure.

The 2732A is in the programming mode when the $\overline{OE}/V_{PP}$ input is at 21V. It is required that a 0.1 μF capacitor be placed across $\overline{OE}/V_{PP}$ and ground to suppress spurious voltage transients which may damage the device. The data to be programmed is applied 8 bits in parallel to the data output pins. The levels required for the address and data inputs are TTL.

When the address and data are stable, a 50 msec, active low, TTL program pulse is applied to the $\overline{CE}$ input. A program pulse must be applied at each address location to be programmed. You can program any location at any time —either individually, sequentially, or at random. The program pulse has a maximum width of 55 msec. The 2732A must not be programmed with a DC signal applied to the $\overline{CE}$ input.

Programming of multiple 2732As in parallel with the same data can be easily accomplished due to the simplicity of the programming requirements. Like inputs of the paralleled 2732As may be connected together when they are programmed with the same data. A low level TTL pulse applied to the $\overline{CE}$ input programs the paralleled 2732As.

Program Inhibit

Programming of multiple 2732As in parallel with different data is also easily accomplished. Except for $\overline{CE}$, all like inputs (including $\overline{OE}$) of the parallel 2732As may be common. A TTL level program pulse applied to a 2732A's $\overline{CE}$ input with $\overline{OE}/V_{PP}$ at 21V will program that 2732A. A high level $\overline{CE}$ input inhibits the other 2732As from being programmed.

Program Verify

A verify should be performed on the programmed bits to determine that they were correctly programmed. The verify is accomplished with $\overline{OE}/V_{PP}$ and $\overline{CE}$ at V_{IL}. Data should be verified t_{DV} after the falling edge of $\overline{CE}$.

System Consideration

The power switching characteristics of HMOS-E EPROMs require careful decoupling of the devices. The supply current, I_{CC}, has three segments that are of interest to the system designer—the standby current level, the active current level, and the transient current peaks that are produced on the falling and rising edges of Chip Enable. The magnitude of these transient current peaks is dependent on the output capacitance loading of the device. The associated transient voltage peaks can be suppressed by complying with Intel's Two-Line Control, as detailed in Intel's Application Note, AP-72, and/or by properly selected decoupling capacitors. It is recommended that a 0.1 μF ceramic capacitor be used on every device between V_{CC} and GND. This should be a high frequency capacitor of low inherent inductance. In addition, a 4.7 μF bulk electrolytic capacitor should be used between V_{CC} and GND for each eight devices. The bulk capacitor should be located near where the power supply is connected to the array. The purpose of the bulk capacitor is to overcome the voltage droop caused by the inductive effects of the PC board-traces.

Finito di stampare nel settembre 1986
presso Lito Velox - Trento
Printed in Italy

Non è l'ennesimo libro sul Vic-20 e il Commodore 64: non insegna a programmare in Basic, ma ad accedere alle risorse fisiche della macchina, per farla dialogare con il mondo esterno, per non lasciarla a sé ma farla intervenire in catene di altre apparecchiature, con compiti di elaborazione e di controllo. Questo volume offre in poco spazio una enorme quantità di informazioni a questo proposito: l'organizzazione hardware della macchina, la gestione dei dispositivi periferici, i file, il chip 6561 di interfaccia video, il linguaggio macchina, le routine del Kernal e la loro utilizzazione, i "cunei" nel sistema operativo e nell'interprete Basic; per concludere con due progetti hardware un convertitore analogico/digitale e un programmatore di EPROM (memorie a sola lettura cancellabili e programmabili). Di particolare interesse la trattazione del convertitore, che permette al Vic o al C64 di effettuare acquisizioni di dati esterni di tipo analogico, per esempio valori di temperatura, tensione, intensità luminosa, e via dicendo, per poi poter effettuare elaborazioni digitali. Questo è essenziale per chi vuole usare il computer con compiti di controllo: in una situazione hobbistica, come l'automazione della casa, o in una situazione professionale (il Vic-20, meglio ancora del C64, si presta ottimamente nel controllo di processi, in laboratorio o nell'industria).

franco muzzio & c. editore

ISBN 88-7021-320-X

L. 25.000 (24.510)

47

Giancarlo Baccolini

Progetti speciali con il Vic-20 e il C64